2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

UNITED STATES NAVY · NAVAL POSTGRADUATE SCHOOL ·

THESIS

Genetic Algorithms for the Development of Real-Time Multi-Heuristic Search Strategies

by
Gary B. Parker
September 1992

Thesis Advisor: Dr. Man-Tak Shing

Approved for public release; distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School — 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION — 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Genetic Algorithms for the Development of Real-Time Multi-Heuristic Search Strategies

12. PERSONAL AUTHOR(S) Parker, Gary Bruce

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis		1992, Sept, 24	131

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Artificial Intelligence, Genetic Algorithms, Search, Path Planning

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Search of an unknown space by a physical agent (such as an autonomous vehicle) is unique in search as the customarily most important goal (to reduce the computation time required to obtain the shortest distance) is not as important as minimal movement. There is a real-time aspect since the agent is actually moving; using energy each step of the way. Having limited energy resources and knowledge of the terrain (only adjacent nodes), the key factor for the physical agent's search algorithm is reduction of steps. Hence, any heuristic that can help keep step count to a minimum must be considered. Korf and Shing addressed this issue in separate works. Both made use of known information about the frontier node's distance from the current node in addition to a heuristic estimating the distance from goal. In this thesis, we present a simple genetics-based method to produce adaptive, efficient multi-heuristic search strategies for the real-time problem. Extensive empirical study shows that this approach produced search strategies with much better performance over existing search algorithms for most terrain types. The methodologies used to develope these improved strategies for our specific case, are also applicable to a multitude of real-time search/optimization problems in the general case.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. Man-Tak Shing	22b. TELEPHONE (Include Area Code) (408) 646-2634 — 22c. OFFICE SYMBOL Code CS/Sh

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Genetic Algorithms for the Development of Real-Time Multi-Heuristic Search Strategies

by
Gary B. Parker
Lieutenant Commander, United States Navy
B. A. Zoology, University of Washington, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

24 September 1992

Author: Gary B Parker
Gary B. Parker

Approved By:
Man-Tak Shing, Thesis Advisor

Yuh-Jeng Lee, Second Reader

Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

Search of an unknown space by a physical agent (such as an autonomous vehicle) is unique in search as the customarily most important goal (to reduce the computation time required to obtain the shortest distance) is not as important as minimal movement. There is a real-time aspect since the agent is actually moving; using energy each step of the way. Having limited energy resources and knowledge of the terrain (only adjacent nodes), the key factor for the physical agent's search algorithm is reduction of steps. Hence, any heuristic that can help keep step count to a minimum must be considered. Korf and Shing addressed this issue in separate works. Both made use of known information about the frontier node's distance from the current node in addition to a heuristic estimating the distance from goal.

In this thesis, we present a simple genetics-based method to produce adaptive, efficient multi-heuristic search strategies for the real-time problem. Extensive empirical study shows that this approach produced search strategies with much better performance over existing search algorithms for most terrain types. The methodologies used to develope these improved strategies for our specific case, are also applicable to a multitude of real-time search/optimization problems in the general case.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Search of an unknown space by a physical agent (such as an autonomous vehicle) is unique in search as the customarily most important goal (to reduce the computation time required to obtain the shortest distance) is not as important as minimal movement. There is a real-time aspect since the agent is actually moving; having limited time to determine its next move and using energy each step of the way. This is in contrast to the traditional problem of search of known space for the shortest path which can be efficiently accomplished by A* search with a good heuristic estimating the distance to goal. The path is found without any movement. Although factors other than the actual distance from start and estimated distance from goal could reduce the number of nodes examined in the traditional problem, these factors usually increase the computational cost per node examined and produce paths that are longer than the shortest path which makes additional heuristics undesirable. Such is not the case for the real-time problem.

The physical agent traversing a terrain in the real-time problem knows only its current position, the goal's position, and whether adjacent and previously adjacent nodes are passable or not. It learns about the terrain only as it moves from node to node examining all nodes adjacent. Information about past nodes, visited or adjacent, can be stored to build up its knowledge base. Computational time to determine the next move is important, as stopping to compute before each move is undesirable. On the other hand, insufficient computations can result in unnecessary steps and wasted energy.

Having limited energy resources and knowledge of the terrain the key factor in the physical agent's search is the reduction of physical steps. In [Pa89], Papadimitrion and Yannakakis showed that the computational problem of deriving optimal search strategies for the real-time problem is PSPACE-complete. Hence, any heuristic that can help keep step count to a minimum must be considered. Korf [Ko90] studied this problem and developed the real-time-A* search, which uses the physical agent's distance from the node ($g(n)$) in addition to the distance from goal heuristic ($h(n)$) to determine the best next

move by minimizing the objective function $f(n) = g(n) + h(n)$ for every adjacent node n. Shing and Mayer [Sh91] developed persistence search which included a persistence factor ($pf = 0$ to 1) to bias the distance from current. The next move is determined by minimizing the objective function $f(n) = pf \times g(n) + h(n)$ for every frontier node n. Experimental results led to the conclusion that the pf could be adjusted to optimize search depending on terrain type and the density of obstacles. Details of these search strategies are in Chapter III.

Extending on these works, we believe a combination of additional heuristics can be beneficial in minimizing physical agent steps. As the number of heuristics increases, it is essential to have some efficient means of assigning bias adjustments to various heuristics to optimize $f(n)$ for different terrain types and densities of obstacles. If the combinatorial explosion required to produce all possible combinations of heuristics is not intractable, the required testing of each to select a best makes this means computationally prohibitive. Since enumeration is probably not possible, some random means of attaining the best combination seems to be the most plausible. DeJong [De75] made clear the advantages of genetic algorithms over purely random selection.

In this thesis, we present a simple genetic algorithm based method to produce adaptive, efficient multi-heuristic search strategies for the real-time problem. Extensive empirical study showed that this approach produced search strategies with much better performance (reduced number of steps without prohibitive computation time) over existing search algorithms for most terrain types. The methodologies used to develope these improved strategies for our specific case, are also applicable to a multitude of real-time search/ optimization problems in the general case.

II. PROBLEM MODEL

A. TERRAIN MATRIX

To best demonstrate the effectiveness of the multi-heuristic search strategies produced by a genetic algorithm, we chose to apply the strategies to random obstacle distributions in the form of a two-dimensional 64x64 grid of squares (nodes). Nodes can be either free or obstacles, movement can be in eight directions through free spaces only. A perimeter surrounding this grid is a solid row/column of obstacles. The distance from a node to its horizontal/vertical neighbor is 1.0; to its diagonal neighbor is $\sqrt{2}$. The total distance traveled from start to goal according to any search scheme is the sum of each of these individual steps. The effectiveness (fitness) of a specific search scheme is the ratio of the shortest path length from start to goal divided by the distance traveled. Given as a percentage, 100 is the best possible; meaning the distance traveled is equivalent to the shortest path. Specific nodes of the grid are be identified by Cartesian coordinates with the left border column being the y axis and the bottom border row being the x axis. The lowest left node is (1,1); the top right is (64,64).

B. DENSITY MATRIX

The 64 by 64 search space grid is divided into 16x16 density blocks, each containing 4x4 nodes and having a specified block density. Block densities range from 0-15. A block density of 9 means that, on average, nine of the block's 16 nodes will be an obstacle (chosen at random). These density blocks are numbered from (0,0) to (15,15) where (0,0) is the bottom left and (15,15) is the top right. Start and Goal positions are specified by density blocks. Most of the block density distributions used will have a start block of (2,2) and a goal block of (13,13). The specific start/goal node is located randomly in that block. See Appendix A.

C. TERRAINS USED

There are ten different density distributions that were used for training and testing. The block densities, once set, remain unchanged from the start of training through testing. Although the block densities remain constant, actual obstacle placement is determined stochastically and changes from run to run. The point is to investigate the adaptability of genetic algorithm to produce the best strategy to direct the search through terrains where the general density distribution is known but actual obstacle placement is not. The first six terrains are considered natural terrains since they closely resemble actual topological conditions. The start density block is always (2,2) unless otherwise stated. The goal density block is always (13,13) unless otherwise stated.

1. Central mountain

The highest density, 15 (denoted as f in Figure 1), is in the center with a gradual decrease towards the lowest density, 1, on the outer edge. Figure 1 shows the density distribution of the terrain in hexadecimal. Transit from start to goal requires a search scheme to find the most efficient way around the mountain.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	1
1	3	5	5	5	5	5	5	5	5	5	5	5	5	3	1
1	3	5	7	7	7	7	7	7	7	7	7	7	5	3	1
1	3	5	7	9	9	9	9	9	9	9	9	7	5	3	1
1	3	5	7	9	b	b	b	b	b	b	9	7	5	3	1
1	3	5	7	9	b	d	d	d	d	b	9	7	5	3	1
1	3	5	7	9	b	d	f	f	d	b	9	7	5	3	1
1	3	5	7	9	b	d	f	f	d	b	9	7	5	3	1
1	3	5	7	9	b	d	d	d	d	b	9	7	5	3	1
1	3	5	7	9	b	b	b	b	b	b	9	7	5	3	1
1	3	5	7	9	9	9	9	9	9	9	9	7	5	3	1
1	3	5	7	7	7	7	7	7	7	7	7	7	5	3	1
1	3	5	5	5	5	5	5	5	5	5	5	5	5	3	1
1	3	3	3	3	3	3	3	3	3	3	3	3	3	3	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 1 : Central Mountain Terrain

2. Single Left Ridge

This terrain has a high density (15) ridge starting from left center moving horizontally out past the grid's midpoint. There is a gradual decrease in density down to 2 as the distance increases from ridge center (Figure 2). Transit through the ridge is not possible.

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
4 4 4 4 4 4 4 4 4 4 4 2 2 2 2 2
6 6 6 6 6 6 6 6 6 6 6 4 2 2 2 2
8 8 8 8 8 8 8 8 8 8 8 6 4 2 2 2
a a a a a a a a a a a 8 6 4 2 2
c c c c c c c c c c c a 8 6 4 2
f f f f f f f f f f f c a 8 6 4
f f f f f f f f f f f c a 8 6 4
c c c c c c c c c c c a 8 6 4 2
a a a a a a a a a a a 8 6 4 2 2
8 8 8 8 8 8 8 8 8 8 8 6 4 2 2 2
6 6 6 6 6 6 6 6 6 6 6 4 2 2 2 2
4 4 4 4 4 4 4 4 4 4 4 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Figure 2 : Single Left Ridge Terrain

3. Single Right Ridge

This terrain is similar to the Single Left Ridge but in the opposite direction (Figure 3). This is a much more difficult situation since the physical agent must move away from the goal to find its best route.

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 4 4 4 4 4 4 4 4 4 4 4
2 2 2 2 4 6 6 6 6 6 6 6 6 6 6 6
2 2 2 4 6 8 8 8 8 8 8 8 8 8 8 8
2 2 4 6 8 a a a a a a a a a a a
2 4 6 8 a c c c c c c c c c c c
4 6 8 a c f f f f f f f f f f f
4 6 8 a c f f f f f f f f f f f
2 4 6 8 a c c c c c c c c c c c
2 2 4 6 8 a a a a a a a a a a a
2 2 2 4 6 8 8 8 8 8 8 8 8 8 8 8
2 2 2 2 4 6 6 6 6 6 6 6 6 6 6 6
2 2 2 2 2 4 4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Figure 3 : Single Right Ridge

4. Double Ridge

This terrain has density areas producing a right ridge on top of a left ridge. Ridge densities are 15 with a valley of 2 in between (Figure 4). An s-shaped path to get from start to goal is required to transit this terrain.

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 4 6 6 6 6 6 6 6 6 6 6 6
2 2 4 6 8 a a a a a a a a a a a
6 6 8 a c f f f f f f f f f f f
4 6 6 8 a c c c c c c c c c c c
4 4 6 6 8 a a a a a a a a a a a
4 4 4 6 6 8 8 8 8 8 8 8 8 8 8 8
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
8 8 8 8 8 8 8 8 8 8 8 6 6 4 4 4
a a a a a a a a a a a 8 6 6 4 4
c c c c c c c c c c c a 8 6 6 4
f f f f f f f f f f f c a 8 6 6
a a a a a a a a a a a 8 6 4 2 2
6 6 6 6 6 6 6 6 6 6 6 4 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Figure 4 : Double Ridge Terrain

5. Single Left Plateau

This terrain is characterized by a large area of high density (10) (dense but passable) starting from left center moving horizontally out past the grids midpoint (Figure 5). The start/goal density blocks are (4,0)/(11,15). A successful transit can consist of either direct passage through the plateau or circumnavigate.

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
a a a a a a a a a a a a a a 8 2
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Figure 5 : Single Left Plateau Terrain

6. Single Left Plateau with Ridges

The same as single left plateau, except it has a ridge of higher density (12) (hard to pass) along the plateaus perimeter (Figure 6). The start/goal density areas are (4,0)/ (11,15). Circumnavigation will usually be the only viable option.

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 2
a a a a a a a a a a a a a a 8 2
c c c c c c c c c c c c c a 8 2
a a a a a a a a a a a a c a 8 2
a a a a a a a a a a a a c a 8 2
a a a a a a a a a a a a c a 8 2
a a a a a a a a a a a a c a 8 2
a a a a a a a a a a a a c a 8 2
a a a a a a a a a a a a c a 8 2
a a a a a a a a a a a a c a 8 2
a a a a a a a a a a a a c a 8 2
c c c c c c c c c c c c c a 8 2
a a a a a a a a a a a a a a 8 2
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Figure 6 : Single Left Plateau with Ridges Terrain

7. Random Terrains

Four different terrains were generated with random block densities set anywhere from 0 to 15 with equal probability. Shown in Appendix B (Figures 30, 31, 32 & 33), these grids were used to test the effectiveness of the different search strategies through unnatural terrains.

III. BACKGROUND WITH DISCUSSION PERTINENT TO MODEL

A. KNOWN TERRAIN SEARCH

A guarantee of optimal path from start to goal is usually the main concern in known terrain search.

1. A* Search

[Ha68] - Widely accepted as the best algorithm for finding the shortest path in a known search space, it uses the actual distance from start and a heuristic estimating the distance from goal. The object of any frontier node is given by the following equation:

$$f(n) = s(n) + h(n) \qquad \text{(Eq 3.1)}$$

where $s(n)$ = the actual distance from start to n, the frontier node, and $h(n)$ = the heuristic estimated distance from n to goal. Although guaranteed to find the optimal path, assuming that the heuristic estimate is always less than the actual, it is not required to minimize the number of nodes examined. Using Euclidean distance as the distance to goal heuristic, A* search provided the shortest path for each terrain used in our experiment to compare the effectiveness of each search scheme.

B. UNKNOWN TERRAIN SEARCH BY A PHYSICAL AGENT

Assuming limited sensory range, the physical agent cannot find the shortest path without excessive moves. Although the shortest path would be nice, more important to the search schemes success is the energy expended/time spent finding a satisfactory path. Distance traveled is of major concern as is time to determine next move (related to this is the computational limitations of the physical agent). The following algorithms have been considered in an attempt to find the best. In all equations, n represents one of the frontier nodes on the frontier list unless otherwise stated.

1. Hill-climb Search

[Wi92] - Depth first search with each move determined by the best (least distance from goal) adjacent frontier node; n In the following equation:

$$f(n) = h(n) \qquad \text{(Eq 3.2)}$$

When no frontier nodes are adjacent to the current, the algorithm backtracks until adjacent frontier nodes are encountered. This search scheme minimizes jumps in search of the best frontier node, but pays the price in extra steps due to unchecked persistence on initially good but eventually poor tracks and the often resultant steps required to backtrack.

2. Real-time-A* Search

[Ko90] - Uses distance from current (actual) and distance from goal (heuristic) to determine best next move. This search only looks at adjacent nodes (frontier and visited). In the following equation n stands for adjacent non obstacle nodes.

$$f(n) = g(n) + h(n) \qquad \text{(Eq 3.3)}$$

$g(n)$ is the actual distance from the current to the adjacent node n. The $h(n)$ is a heuristic predicting the distance from n to the goal. Initially, $h(n)$ is calculated by using Euclidean distance in our example. The algorithm picks the adjacent node with the best $f(n)$. Before moving, the value of the $f(n)$ of the second best adjacent node is stored in the current node. This stored value will, in future computations, be the node's $h(n)$. This value remains constant until the node is visited again. This well conceived search scheme requires minimal computations and memory, but is subject to wasted moves when drawn into local traps.

3. Best-first Search

[Wi92] (modified for physical agent) - Uses only the distance from goal heuristic (Euclidean distance) to select the next move.

$$f(n) = h(n) \qquad \text{(Eq 3.4)}$$

Once selected, it uses the shortest path through visited nodes (known search space) to travel the distance from the current node to the selected frontier node. Although, after each move it is at the best known location, the cost of getting there can be expensive. In worst case situations it can end up jumping large distances back and forth while zeroing in on the goal.

4. Persistence Search

[Sh91] - Similar in concept to Real-time-A*, it uses the distance from goal heuristic and a weighted distance from current to determine its next move. Unlike Real-time-A*, it makes more use of known information; resulting in better moves, but decreased computational efficiency. The distance from goal is Euclidean. The distance from current to frontier is the shortest path through visited nodes as in Best-first, but this distance is weighted and used in determining the next move. The object of a frontier node n is given by

$$f(n) = pf \times g(n) + h(n) \qquad \text{(Eq 3.5)}$$

where $g(n)$ = shortest distance from current position to n through visited nodes, $h(n)$ = Euclidean distance from n to goal. A persistence factor (pf = 0.0 to 1.0) is added to vary the relative contribution of each of the heuristics to the determination of next move. Distance from current, assumed to be always less pertinent, can be reduced in importance in comparison to distance from goal.

C. GENETIC ALGORITHMS

Genetic algorithms, developed by John Holland [Go89] and his associates, are based on the laws of natural selection and survival of the fittest. Subjecting a population (animals, search schemes, etc.) to environments where fitness for survival is required, individuals best suited for survival will flourish and reproduce while individuals lacking the diversity required to continue in all possible environments will discontinue.

The key to the success of a population is its robustness [Go89]. An individual, and therefore a population, is made up of traits which are derived from specific genes in the individuals chromosome [St77]. Applicable traits in the animal world are weight, height,

leg length, neck length, etc. A combination of these traits describe an individual. Extremes in any one trait usually means more specification and added survivability in a limited range of environments, whereas moderation in traits means added adaptability for diverse environments. The key is to find the balance of these two in a population to give it proper robustness. Example: the giraffe can afford to be specifically designed for reaching (long neck and legs) because it doesn't face diversities in environment that would require escape through low canopy jungles. It is perfectly adapted for life on the plains with occasional trees.

Similarly, search strategies can be very specialized in simple environments. Search through a low density (of obstacles) terrain can be successfully accomplished with efficiency and consistency by a simple hill climb algorithm (only one trait, distance to goal of adjacent nodes, is important). Search problems involving more complicated and diversified solutions require the proper balance of traits (heuristics) to solve. Simple direct "hill climbing" approaches can often result in searching locally optimal blind alleys.

One possible means of developing the balance of traits required to avoid getting stuck in the local minimum is to enumerate all possible combinations. This would most assuredly find the optimum, but in many problems the combinatorial explosion of possibilities make this method prohibitive. Purely random combination of trials is a possibility that seems to avoid both the local minima and the combinatorial explosion problems. But on further examination, it suffers the same drawbacks as enumeration, in that there are only a limited number of trials possible whether you look at them in order or at random. Genetic algorithms use randomness as a tool in a direct search for the optima. Promising potential solutions can be searched in parallel while feedback information is used to select the next partially random strategy. The results, as evaluated by DeJong [De75], show the superiority of genetic algorithms over purely random.

The basic genetic algorithm makes use of a population of individuals (usually binary strings of fixed length) that are made up of the traits pertinent to the problem (traits are usually represented by a fixed number of the bits in specific locations). Three genetic

operators are used to transform the original (randomly generated) population into an optimal one: selection, crossover, and mutation.

The Fitness of an individual of the population is established by some form of evaluation function. One scheme is to compare each to a known optimum, assigning higher fitness to ones approaching the optima. This evaluation can also be averaged over some set number of trials (cycles) for each individual and then assigned as the fitness before forming the next generation.

Each new generation of the population is formed by stochastically selecting individuals from the prior population. Higher fitness individuals have a higher chance of being selected. Reproduction is performed by randomly pairing selected individuals for crossover and mutation.

Crossover is performed at a random point in the binary string. The two selected strings interchange their tail sections at the crossover point to from two new individuals. The crossover point can be anywhere from 0 to the last bit. For example, let the two selected strings be 00000000 and 11111111, and let 5 be the crossover point. Then the crossover operation will produce the new strings 00000111 and 11111000. In general, crossover forms two new individuals with one hopefully having all the best from its two parents.

Mutation is a bit by bit operator that takes each individual and randomly (with a specified probability) decides if each bit will be changed? For example, changing the second bit of the string 00000111 by mutation will result in the new string 01000111. This genetic operator, as in nature, ensures that populations maintain adaptability even when specialization is the rule. An extremely high mutation probability regresses the genetic algorithms to a uniform randomly distributed population, a very low one reduces the populations adaptability. A happy medium seems to be in the 0.01 to 0.001 range for probability of individual bit mutation.

A myriad of variations are possible to improve the performance and robustness of the genetic algorithms. For the purpose of our research, these were found to be unnecessary, and will not be covered in this discussion.

IV. FACTORS RELEVANT TO SEARCH

A. DISTANCE FROM START

This is usually the actual shortest path from the start node to the considered frontier. Currently believed to be useless in a real-time environment, it should be selectively eliminated by natural selection as the genetic algorithm trains. For our implementation, it is approximated by computing the Euclidean distance from start to frontier. It may be significant in some of the more complex terrains that require a switch back.

B. DISTANCE FROM CURRENT

The distance from the current node to the frontier node; important in Real-Time-A* and Persistence Search to determine if backtracking is worth the steps required. It is the actual distance computed as the actual steps required to move from the current node to the frontier.

C. DISTANCE FROM GOAL

The Euclidean distance from the current node to the goal node. This heuristic is usually considered important in any search. It is used in combination with "distance from current" for Persistence Search, and by itself for Best-first Search.

D. CROWDING

The crowding parameters, crowding sides and crowding diagonals, are an attempt to assist the physical object in avoiding areas of increased obstacle density. This reduces exploration of paths through high density areas, favoring the safer path of increased options available in the open space. The parameters are separated in case one is more appropriate than the other. Both would be much more effective without the self imposed constraint of physical object perception only being adjacent nodes. If all nodes adjacent to the frontier node could be seen, these factors importance would increase significantly.

1. Crowding sides

This heuristic examines the frontier node's known horizontal/vertical neighbors to count the number of obstacles. Nodes with more known obstacle neighbors are less desirable. The minimum value is 0 and 4 is the maximum.

2. Crowding diagonals

This is similar to the previous parameter with the count being made of the frontier node's diagonal neighbors.

E. MOVE AWAY FACTOR

It attempts to continually reduce the search space by reducing desirability of nodes that increase the x and/or y difference between the current and goal nodes. Increasing the x or y distance counts as 2, increasing both counts as 4, and no increase results in the heuristic having a value of 0.

F. MOMENTUM

This heuristic attempts to avoid zigzag by making forward (in relation to last move) nodes the most desirable. It should be useful in valley/ridge terrains where the best path is straight through the valley. By maintaining momentum, the physical object avoids steps wasted in popping in and out of each crevice which has nodes closer to the goal. Straight ahead movement results in a value of 0, a 45° shift makes it 1, a 90° shift is 2, and a 135° shift or non-adjacent move results in a value of 3 (making only the adjacent nodes subject to change after a move).

V. PROGRAM DEVELOPMENT

A. DATA STRUCTURES

1. Node structure

The 64x64 grid is internally represented as a 66x66 two dimensional array (the perimeter nodes are all marked as obstacles) made up of pointers to node records. The records store information pertinent to terrain, search (heuristics), graphic display, and pointers to other node records (used in the program for various dynamic structures). The heuristic values stored include distance from start, distance from goal, distance from current, crowding sides, crowding diagonals and subtotal. No other node records are used in the program; other structures requiring nodes are set up using pointers to these records.

2. Population structure

A 32 member array of individual records makes up the population. Each stores the individual's fitness and its chromosome which contains biases for each search parameter. The chromosome is a 32 bit unsigned integer; subdivided into eight four-bit unsigned integers, it holds up to eight heuristic bias factors with a range from 0 to 15.

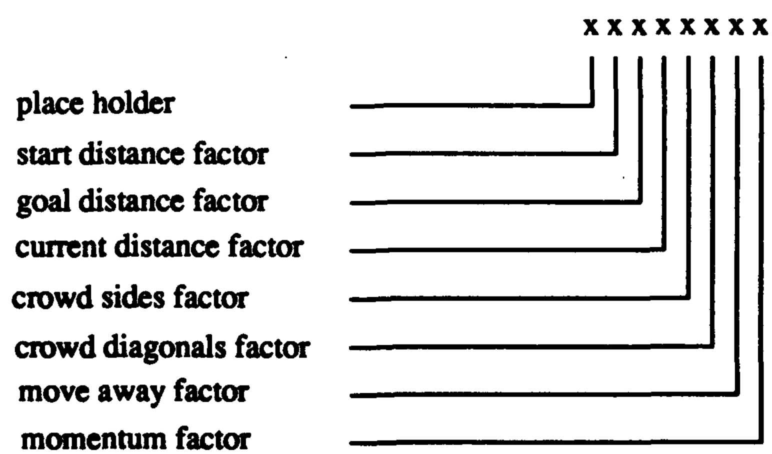

Figure 7 : Individual Chromosome Structure

3. Frontier Heap

Implemented as an array of pointers to node records. Functions to manipulate this min-heap are in theap.c.

4. Mate Heap

An array of 31 integers. The first is a count of the heaps members, the other 30 are chosen at random in the range of 0 to the sum of all individual fitnesses. They are used to randomly select individuals for reproduction. Functions to manipulate this heap are in eheap.c.

B. SEARCH ALGORITHMS

Algorithms used for this analysis, with appropriate modifications, are covered in this section. Each node, represented as a record, has a number of fields that are used to store needed information. When a function operating on a specific node v is used, a read or write to the appropriate field takes place. For example: in the A* algorithm the following calculation takes place; $f(v) = s(v) + h(v)$. The value $s(v)$ calculated earlier was stored in v.s (the nodes dist_from_start field), $h(v)$ is calculated and stored in v.h (the nodes dist_from_goal field), and $f(v)$ is stored in v.f (the nodes subtotal field).

1. A* Search

This search (Figure 8) will find the shortest path from start to goal if a path exists. The heuristic estimating distance to goal ($h(v)$) is a lower bound of the actual cost of the optimal path from v to the goal.

2. Hill-Climb Search

This search (Figure 9) always moves forward until there is no where to go. It then back-tracks the way it came until a move is possible. It is a depth-first search with a heuristic to determine the best move to advance it to the goal.

a_star_search

```
current := start
while current != goal do
    for all nodes, v, adjacent to current do
        if UNTOUCHED
            s(v) = current.s + distance(current, v)      /* Euclidean */
            f(v) = s(v) + h(v) /* h(v) is the Euclidean dist to goal */
            add to frontier heap
        elsif FRONTIER
            if s(v) > current.s + distance(current, v)
                update s(v) and f(v)
                update position in frontier heap
            endif
        endif
        if frontier_heap is empty
            return BIG_NUMBER /* there is no path from start to goal */
        endif
    end for loop
    current := top(frontier_heap)
end while loop
return goal.s
```

Figure 8 : A* Search Algorithm

hill_climb_search

```
(1)   current := start
(2)   while current != goal do
(3)       best := dummy_node           /* f(dummy_node) = BIG_NUMBER */
(4)       for all nodes, v, adjacent to current do
(5)           if UNTOUCHED
(6)               f(v) := distance(v, goal) /* the Euclidean dist to goal */
(7)               mark v as FRONTIER
(8)           endif
(9)           if FRONTIER and f(best) > f(v)
(10)              best := v
(11)          endif
(12)      end for loop
(13)      previous_current := current
(14)      if best != dummy_node
(15)          current := best
(16)          current.predicessor := previous_current
(17)      elsif current != start
(18)          current := current.predicessor
(19)      else
(20)          return BIG_NUMBER
(21)      endif
(22)      current.dist_traveled :=
          previous_current.dist_traveled + distance(current, previous_current)
(23)  end while loop
(24)  return goal.dist_traveled
```

Figure 9 : Hill Climb Search Algorithm

3. Real-Time-A* Search

This algorithm (Figure 10) is in accordance with Korf's description [Ko90]. For our implementation, the node array was used to store the h value since it was already in place, negating the necessity for a hash table.

4. Best-First Search

This search (Figure 11) always goes to the best (minimum h(v)) node regardless of its distance from the current node. It is possible to implement as a specific case of the multi-heuristic search (Figure 13).

5. Persistence Search

Shown in Figure 12, *gf* + *hf* are intended to effectively replace/descretize/expand the persistence factor, *pf*, in the original work ([Sh91] equation 3.5). *pf* can have any value between 0.0 and 1.0. We found that an infinite range of possibilities for this factor was not required. A descrete, yet sufficient, span can be obtained by setting *gf* and *hf* to any number of possibilities where $gf \leq hf$. Setting *hf* to 15 and incrementing *gf* from 0 to 15 gives us the equivalent of a 0.0 to 1.0 range incriminated by 0.067.

$$f(v) = gf \times g(v) + hf \times h(v) \qquad \text{(Eq 5.1)}$$

There is also now the expanded capability of having the g(v) be the more important factor in the search ($gf > hf$). This search can also be implemented as a specific case of the multi-heuristic search.

real_time_astar_search

```
(1)   current := start
(2)   best := dummy_node
(3)   second_best := dummy_node              /* f(dummy_node) = BIG_NUMBER */
(4)   while current != goal do
(5)       for all nodes, v, adjacent to current do
(6)           if UNTOUCHED
(7)               h(v) := distance(v, goal) /* Euclidean */
              /* else h(v) is already set */
(8)           endif
(9)           g(v) := distance(current, v)
(10)          f(v) := g(v) + h(v)
(11)          if best.f > f(v)
(12)              second_best := best
(13)              best := v
(14)          elsif second_best.f > f(v)
(15)              second_best := v
(16)          endif
(17)      end for loop
(18)      previous_current := current
(19)      current := best
(20)      previous_current.h := second_best.f
(21)      current.dist_traveled :=
          previous_current.dist_traveled + distance(current, previous_current)
(22)  end while loop
(23)  return goal.dist_traveled
```

Figure 10 : Real Time A* Search Algorithm

best_first_search

```
(1)   current := start
(2)   while current != goal do
(3)       for all nodes adjacent to current do
(4)           if UNTOUCHED
(5)               h(v) := distance(node, goal) /* Euclidean */
(6)               add to frontier_heap
(7)           endif
(8)       end for loop
(9)       if empty(frontier_heap)
(10)          return BIG_NUMBER           /* no solution */
(11)      endif
(12)      v.dist_traveled := current.dist_traveled + g(v)
          where g(v) is the shortest distance through known paths from
          current to frontier node.
(13)      previous_current := current
(14)      current := top(frontier_heap) /* minimum h(v) */
(15)  end while loop
(16)  return goal.dist_traveled
```

Figure 11 : Best First Search

persistence_search

```
(1)   current := start
(2)   while current != goal do
(3)       for all nodes adjacent to current do
(4)           if UNTOUCHED
(5)               h(v) := distance(node, goal) /* Euclidean */
(6)               add to frontier_heap
(7)           endif
(8)       end for loop
(9)       if empty(frontier_heap)
(10)          return BIG_NUMBER     /* no solution */
(11)      endif
(12)      find the frontier node, v, that minimizes the equation:
          f(v) := gf * g(v) + hf * h(v)    where g(v) is the shortest distance through
          known paths from current to frontier node. gf and hf, set before search,
          are bias factors used to vary the relative importance of g(v) and h(v).
          They can have a value from 0 to 15.
(13)      v.dist_traveled := current.dist_traveled + g(v)
(14)      current:= v
(15)      remove current from frontier_heap and update
(16)  end while loop
(17)  return goal.dist_traveled
```

Figure 12 : Persistence Search Algorithm

6. Multi_heuristic Search

This is the general algorithm (Figure 13) enstantiated in our case to handle five stable_heuristics and two unstable_heuristics. Stable_heuristics being ones that have values that will not change if more than two steps away from the current node. They include Euclidean distance from goal (hg), Euclidean distance from start (hs), crowd sides (hcs), crowd diagonals (hcd), and momentum (hm). The subtotal $fs(v)$ is calculated using these functions multiplied by their respective bias factor and stored in v.subtotal.

$$fs(v) = hgf \times hg(v) + hsf \times hs(v) + hcsf \times hcs(v) + hcdf \times hcd(v) + hmf \times hm(v) \quad \text{(Eq 5.2)}$$

Unstable_heuristics have values that are liable to change as the current node changes. Examples in our case: distance from current (hdc) and move away (hma). The algorithm minimizes equation 5.3 using the efficient "branch-and-bound" search through known (visited) nodes described in section 4.3 of [Sh91].

$$f(v) = fs(v) + hdcf \times hdc(v) + hmaf \times hma(v) \quad \text{(Eq 5.3)}$$

The hsf, hgf, $hdcf$, $hcsf$, $hcdf$, $hmaf$, and hmf are bias factors that correspond with the individual chromosome's lower 28 bits which are set during training. The highest four bits are, in our implementation, a place holder for future additional heuristics since only seven applicable heuristics were identified. Note that the Best-first and Persistence Search could be implemented as special cases of the multi-heuristic search algorithm. Best-first uses an individual chromosome input of 00100000 (the third factor being hgf). Persistence Search uses an individual chromosome input of 00xy0000 with x and y varying from 0 to 15 (fourth factor being $hdcf$).

multi_heuristic_search

```
(1)   current := start
(2)   while current != goal do
(3)       for all nodes v within 2 moves of current do
(4)           if adjacent and UNTOUCHED
(5)               v.subtotal := inner_product(stable_heuristics * respective_biases)
(6)               add v to frontier_heap        /* min subtotal node on top */
(7)           elsif FRONTIER
(8)               if any stable_heuristics of v have changed
(9)                   v.subtotal := v.subtotal + adjustment
(10)                  update position in frontier_heap
(11)              endif
(12)          end if
(13)      end for loop
(14)      if empty (frontier_heap)
(15)          return BIG_NUMBER     /* no solution */
(16)      endif
(17)      find frontier node, v, that minimizes
          f(v) = v.subtotal + inner_product(unstable_heuristics * respective_biases)
(18)      v.dist_traveled := current.dist_traveled + g(v)
          where g(v) is the shortest distance through known paths from current to
          frontier node.
(19)      current := v        /* and remove v from heap */
(20)  end while loop
(21)  return goal.dist_traveled
```

Figure 13 : Multi Heuristic Search Algorithm

C. GENETIC ALGORITHM

The task of the genetic algorithm is to find the combination of the seven bias factors that will result in the optimum search scheme. The values of these seven bias factors are stored in a single individuals chromosome. Application of genetic operators to a population (32 in our case) of these individuals will, after numerous iterations, produce our desired optimal individual.

The genetic algorithm, described in this section, is invoked during training after some predetermined number of cycles (making up one generation). The input population will have a fitness value (ability to get through the terrain) assigned to each of it's 32 individuals (details of this process are described in the next chapter). This fitness value and the individual's chromosomal make-up are required by the genetic algorithm.

Our algorithm (Figure 16) makes use of the three genetic operators: selection, crossover, and mutation. The implementation is similar to the algorithm presented in chapter one of the text by Goldberg, [Go89], with the additions of allowing the best two individuals to go unchanged and an average of one out of seven of the remaining not going through crossover. The result is similar to De Jong's R3 elitist model [De75]. Examples of our crossover implementation are detailed in Figures 14 & 15.

```
Alleles are represented in hexidecimal
Before allele crossover:          55555555 / 88888888
Randomly picked crossover allele position is 3  (4th allele)
After allele crossover:           55558888 / 88885555
```

Figure 14 : Allele Crossover Example

```
The 4th allele is expanded out into binary representation
Before bit crossover:          555 0101 8888 / 888 1000 5555
Randomly picked crossover position between bits is 2
After bit crossover:           555 1001 8888 / 888 0100 5555
```

Figure 15 : Bit Crossover Example

genetic_algorithm

(input is a population of individuals)

```
(1)   total_fitness := all individual fitnesses added together
(2)   select 32 individuals as follows                /* selection */
(3)       best := individual with the highest fitness
(4)       second_best := individual with the second highest fitness
          /*second_best must be distinct from best */
(5)       stochastically select 30 individuals with higher fitness individuals having
          the greatest chance of selection
(6)   end selection
(7)   create new_population with these 32 individuals
      pair individuals in such a way that it is unlikely that an individual is paired
      with itself; pair best with second_best
(8)   for each individual pair, except best and second best, do
(9)       randomly pick crossover allele position                /* crossover */
(10)      if not 0    /* 0 means no crossover */
(11)          exchange all alleles after the crossover allele
(12)          randomly pick crossover position between bits of selected allele
(13)          if not 0 or 4    /* 0 or 4 means crossover does not breakup the allele */
(14)              exchange bits after the crossover position between bits
(15)          endif
(16)      endif
(17)      for each gene of the individual                        /* mutation */
(18)          invert bit if random < prob of mutate
(19)      end for loop
(20)  end for loop
(21)  add best + second_best to new_population as individuals 0 & 1 respectively.
(22)  return new_population
```

Figure 16 : Genetic Algorithm

VI. TRAINING

Training of the population is analogous to selectively breeding a random group of asexual organisms to obtain superior capability in a specific area. The capability you wish to optimize is transit from start to goal in the least number of steps. The specific area is a specific terrain layout where you have an idea about general areas of density, but have no information about the location of specific obstacles.

The first step is to generate a series of specific terrains from your general idea of the densities. This can be done by placing obstacles in each area if a randomly generated number is less than the specified density. In our implementation, we simply loop through the 64x64 node array assigning each nodes state to OBSTACLE if the random number is less than the density value of the corresponding density block. The second step is to generate a population of 32 individuals giving them randomly generated chromosomes. Now the training begins (Figure 17). In all our work, we used 1000 generations with five cycles (trials) per generation.

The returned population evolves through the trials of 5000 terrains. One of the individuals of this population is likely to have a chromosome that approximates the optimum combination of bias factors. Identification of this individual is accomplished during testing.

training

(1) for the number of generations do

(2) for the number of cycles do

(3) loop until a successful A* search

(4) create a terrain from the density_array

(5) shortest_path := A* search

(6) end until loop

(7) run each individual through the terrain accumulating its fitness_sum by comparing its actual path to the shortest path

(8) end for loop

(9) compute each individual's average fitness from fitness_sum and number of cycles

(10) apply the genetic algorithm to the population

(11) end for loop

(12) return a trained population.

Figure 17 : Training Algorithm

VII. TESTING

Testing of the trained populations was performed by comparing the search conducted by the best individual in each population to searches accomplished using Hill-climbing, Best-first, Real-time-A*, and Persistence search. The following equation was used to compute fitness for all search schemes:

$$fitness = integer(((shortestpath) + (actualpath)) \times 100) \qquad \text{(Eq 7.1)}$$

Each search scheme was tested on 500 distinct terrains produced using the corresponding density matrix.

Before testing, the best of each population was chosen by running the population through 50 distinct terrains. The individual with the highest fitness was chosen to represent the GA-trained population. The best values for distance from goal and distance from current bias factors for the Persistence search were determined by running 32 combinations (chromosomes of 00f00000 to 00ff0000 and 000f0000 to 00ff0000) through 50 distinct terrains. From this, the best combinations of the two factors was used to represent Persistence search.

The GA-produced best individual, Persistence best, Hill-climb, Real-time A*, and Best-first schemes were then all used to find a path in the 500 separate terrains. Average fitnesses over the 500 were assigned and a comparison of these fitnesses is presented in the results.

VIII. EXPERIMENTAL RESULTS

The fitness of each search scheme in these results is the number of its required steps divided by the minimum steps possible, averaged over the 500 terrains used for testing. Fitness is presented as a percentage, with a 100% search scheme being one that can, on the average, search a terrain type in the minimum steps possible. In general, the easier the density layout of the terrain, the higher the fitness will be.

A. NATURAL TERRAINS

A graph comparing the fitness of applicable search schemes is presented for each natural terrain density layout (Figures 18 - 23). The following discussion is pertinent to each of these comparisons.

1. Central Mountain

This graph (Figure 18) shows that this terrain is only moderately hard for all the search schemes. Persistence search with a distance to current factor (gf) of 15 and a distance to goal factor (hf) of 11 (gf/hf = 15/11) was the best of the conventional search methods. The genetic algorithm produced an individual with chromosomal make-up of f00732b9 (see figure 7, page 16 for breakdown) which performed 1.20 times better than the best conventional. Driven more to the goal by the move-away heuristic than distance to goal, this scheme was better equipped to avoid the congestion of the central mass.

2. Single Left Ridge

Overall this terrain was a little harder than the Central Mountain but was still handled moderately well by all search strategies (Figure 19). The best conventional was again persistence search using a gf/hf ratio of 15/6. The genetic algorithm scheme (f00828ff) had a fitness 1.16 times as good as the best Persistence and 1.28 times better than the next competitor (Hill-climbing).

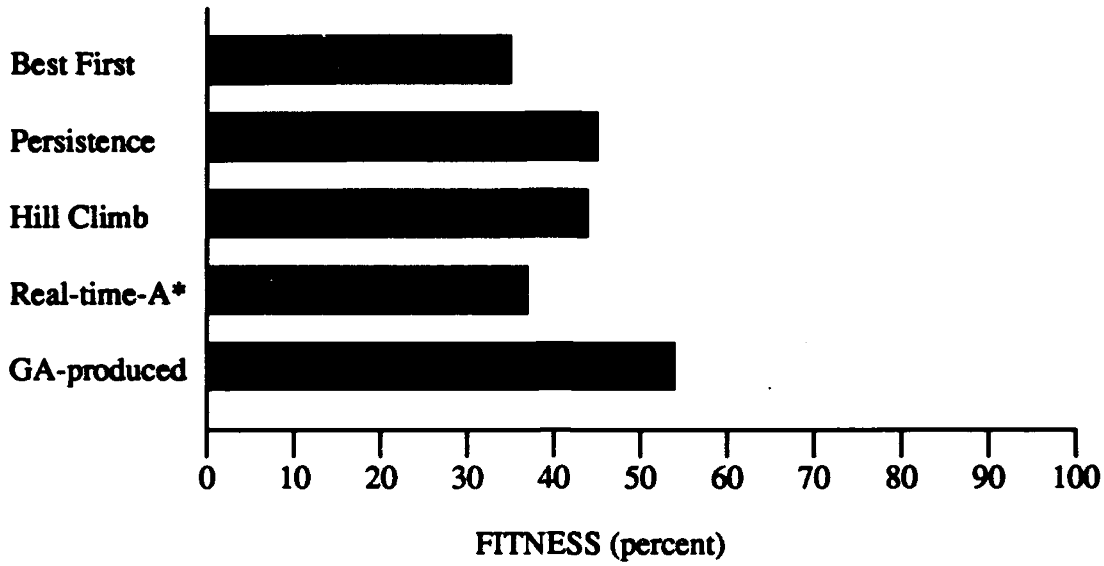

Figure 18 : Central Mountain Results

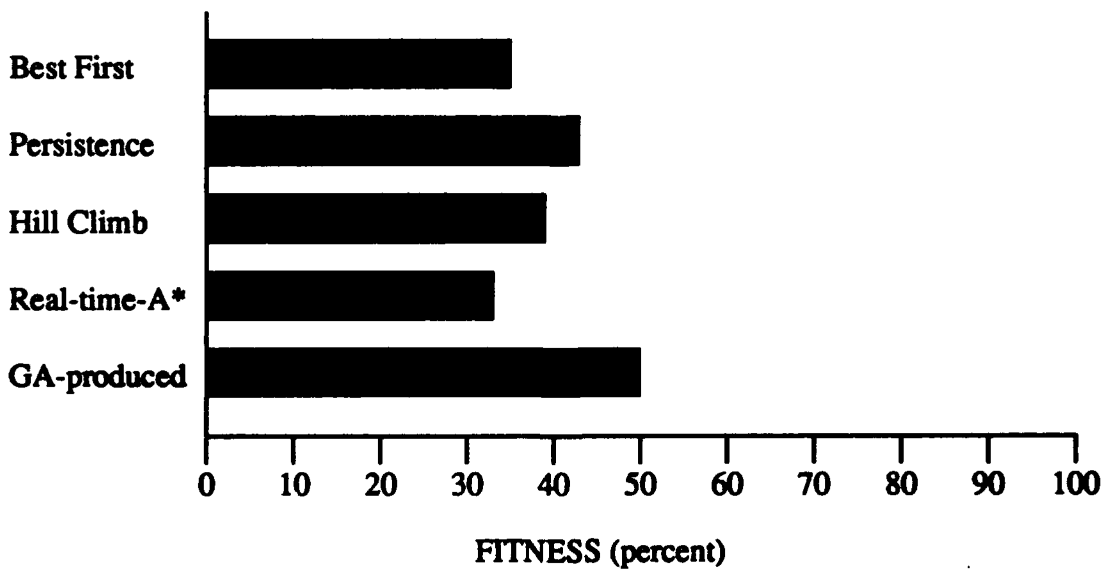

Figure 19 : Single Left Ridge Results

3. Single Right Ridge

As could be predicted, this was a hard problem (Figure 20). The normal search schemes tend to spend a lot of time searching every possible route that was most direct to the goal. They would get stuck under the ridge with no way around, except back the same way they came. The best of these was Hill-climbing since it probably doesn't waste a lot of steps backtracking. The best Persistence search had a gf/hf ratio of 2/15 showing it's favoritism for a no backtrack approach. The search scheme produced by the genetic algorithm was superior to all by a multiplication factor of 1.26. Its chromosomal make-up was f00c2ca8. This scheme considers distance to goal to be not significant. It instead uses move away factor as the drive toward the goal. As the Persistence search with its gf/hf ratio of 2/15 the genetic algorithm produced scheme considered the amount of backtracking a major factor.

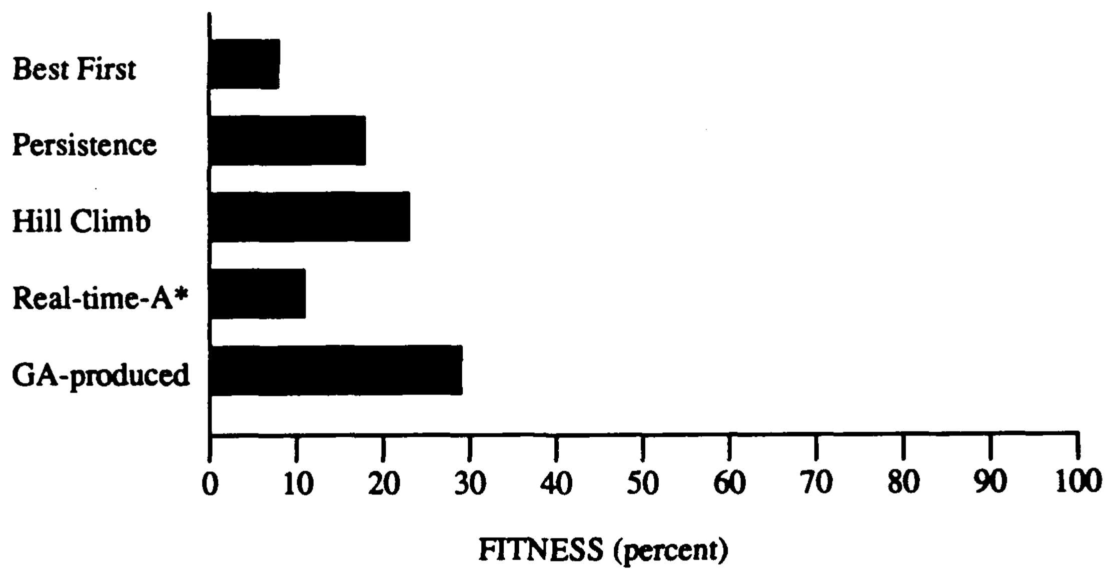

Figure 20 : Single Right Ridge Results

4. Double Ridge

This terrain was a very difficult problem (Figure 21). Requiring negotiation around two ridges which involved a switch-back away from the goal, none of the search schemes were over 20% fitness. The average of the five schemes was 10%. The best Persistence, with a gh/hf ratio of 6/15, was roughly equivalent to Hill-climbing. The genetic algorithm generated scheme with a chromosomal make-up of f83b19bc was the best strategy with a fitness 1.21 times better than Persistence. Here is an example where distance to start was of significance; probably helpful in influencing the search to make the switch-back away from the goal. Move away factor was a major influence in striving toward the goal, backtracking was determined to be non-productive, but maintaining momentum was found to be important. It's interesting to note that diagonal crowding was considered more important than side crowding (no explanation). The complexity of this scheme with the subtle interaction between these differing bias factors helps to confirm the necessity of a genetic algorithm to sort them out.

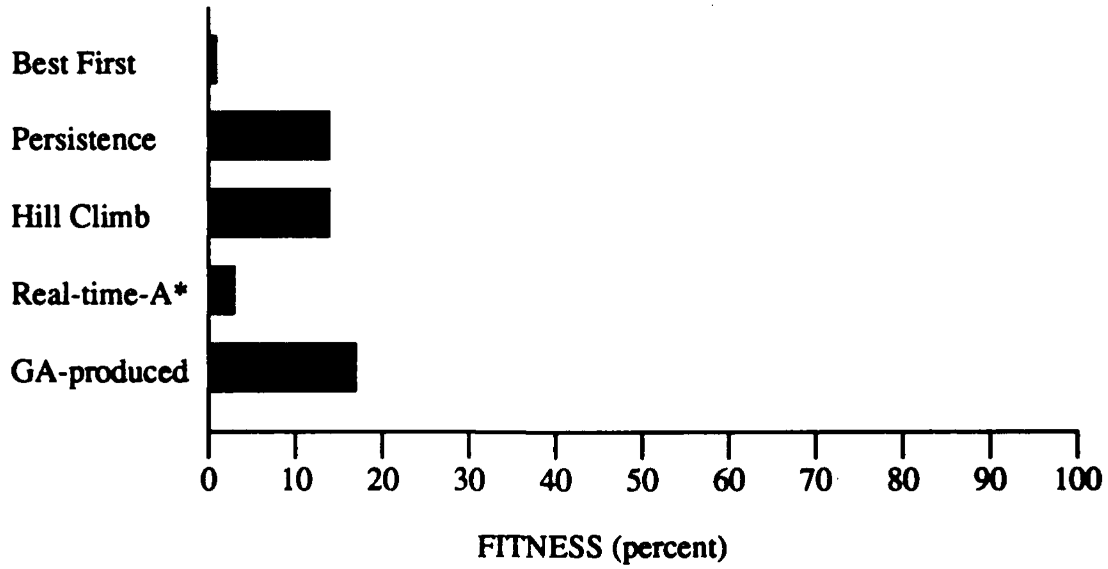

Figure 21 : Double Right Left Ridge Results

5. Single Left Plateau

This terrain was slightly easier for all the search strategies (Figure 22) although it presented a unique problem. The through the plateau route is possible but requires numerous explorations. Circumnavigating the plateau saves exploration steps but costs in the distance required. Since each of the 500 terrains had varying obstacle placement, we suspect sometimes it was best to transit through and other times better to go around. Since no general path was consistently optimal, the genetic algorithm had to develope a scheme that was equally effective for both routes or concentrate on perfecting one. In either case, its performance was again superior by a significant margin (multiplicative factor of 1.19). The resultant chromosomal make-up was f05e884f. The next best was Persistence with a gf/hf ratio of 11/15.

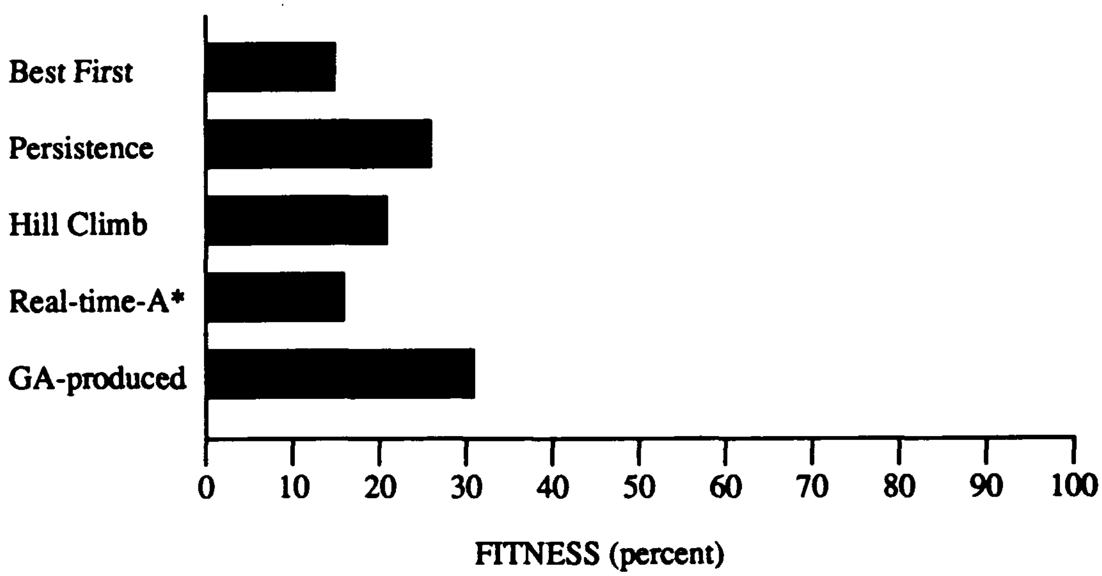

Figure 22 : Single Left Plateau Results

6. Single Left Plateau With Ridges

This terrain adds topological characteristics that favor circumnavigation as a search strategy. The genetic algorithm produced scheme, with a chromosomal make-up of f07c033d, was the best by a multiplication factor of 1.17 over the next best competitor (Figure 23). Momentum being the most important factor, it probably helped keep the search moving horizontally until clear of the plateau. Persistence was again the second best with a gh/hf ratio of 11/15.

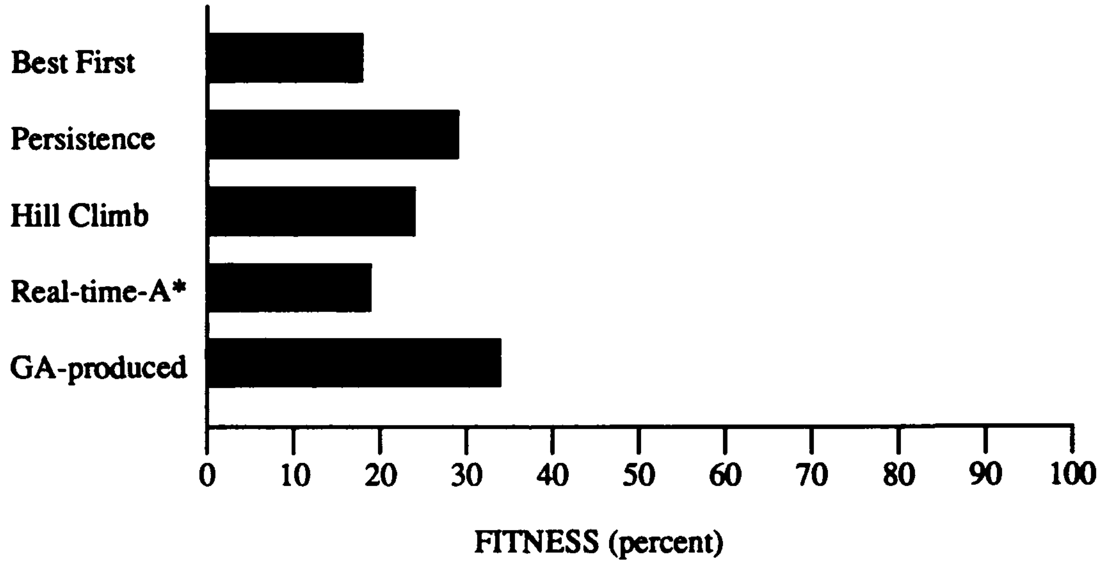

Figure 23 : Single Left Plateau With Ridges Results

7. General Comment

The genetic algorithm was extremely successful in producing the best search strategies for all natural terrains.

B. RANDOM TERRAINS

Although the genetic algorithm produced search schemes where clearly superior for the natural terrains, we wanted to test their viability on randomly generated terrains.

1. Random One / Random Two / Random Three

The results from these three terrains showed that the search heuristics produced by genetic algorithms was of minimal value (Figures 24 to 24). These were all simple problems with the average fitness for all the search schemes being 64%. Fitness varied little between search strategies with a maximum of a 8% difference between the best and the worst. The genetic algorithm produced scheme was 1.02 (Random One), 1.01 (Random Two), and 1.03 (Random Three) times as good as the best conventional search strategy. The Random One persistence gf/hf ratio was 15/11; the genetic algorithm produced chromosome was f1e90234. The Random Two gf/hf = 15/4; GA-produced chromosome = f0b947c1. The Random Three gh/hf = 15/5; GA-produced chromosome = f1f73351.

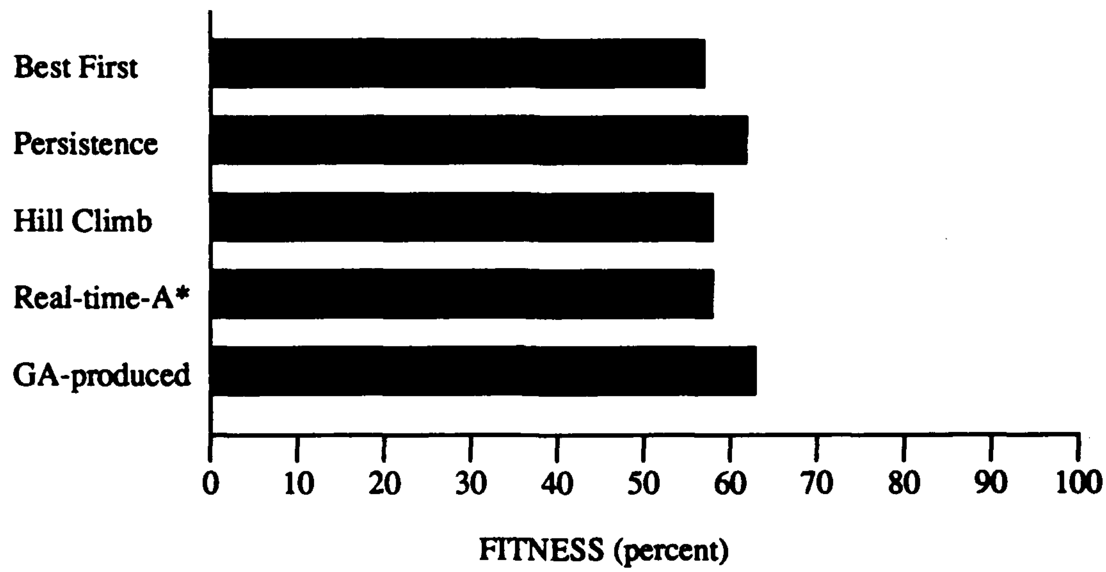

Figure 24 : Random One Results

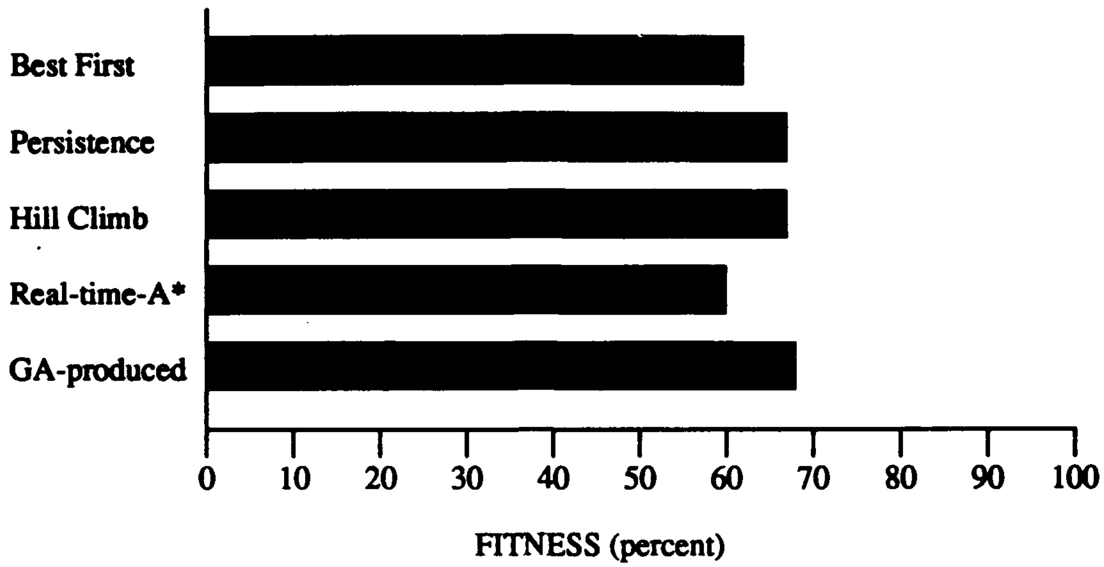

Figure 25 : Random Two Results

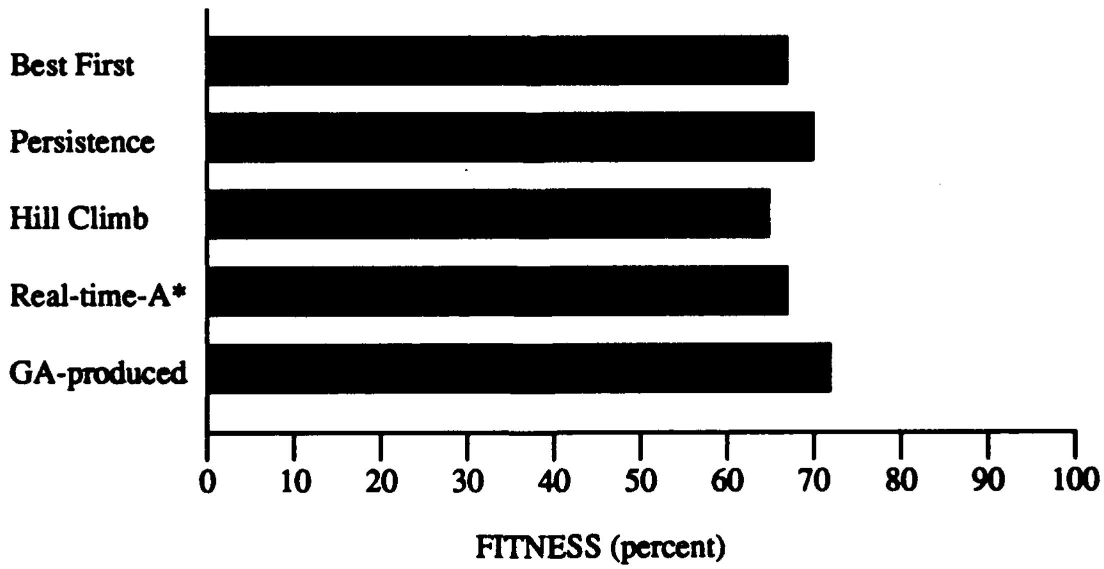

Figure 26 : Random Three Results

2. Random Four

This terrain was significantly harder than the other three randomly generated terrains which can be observed by the low performance of the search strategies. The average fitness of all strategies was 36% (Figure 24). The difficulty probably comes from the encapsulation of the goal. Examining figure 33, page 46, we can see that the goal is blocked by mostly high density blocks from (11,15) down to (11,10) across to (15,10). The only passible blocks are (15,10) and (11,12). Neither of which are a direct route, necessitating significant exploration. The genetic algorithm produced (f0b51535) scheme was 1.10 times better than the next best which was a persistence strategy with a gf/hf ratio of 14/15. This again seems to suggest that the genetic algorithm is only required when the problem is hard.

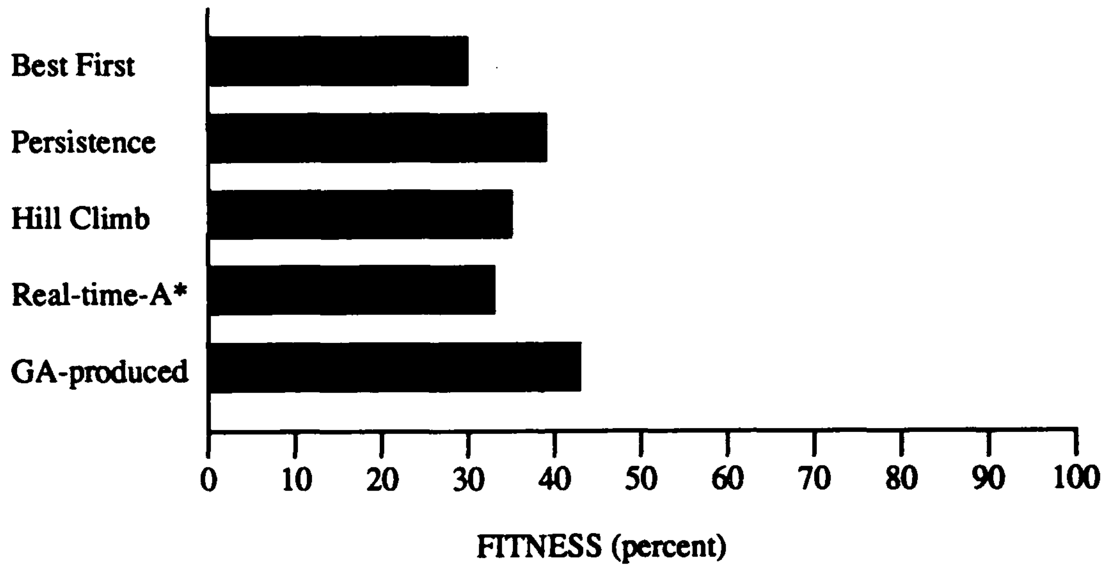

Figure 27 : Random Four Results

3. General Comments

It is difficult to improve on the simple search strategies when the terrain is of low complexity. As suggested by DeJong [De92] the genetic algorithm can only optimize to a certain point (dependent on implementation) before reaching a state of dynamic equilibrium. The first three random terrains were of insufficient complexity to allow the genetic algorithm to convincingly surpass all conventional search schemes. It was however, in all cases, better than the best conventional ones.

C. GENERAL COMMENTS

In all cases, although the genetic algorithm produced strategy was always as least as good as the next best, it was not a substantial improvement over Persistence search unless the terrain was natural. Only in the most complex of the four random terrains did the genetic algorithm produced scheme really excel. This seems to suggest that the additional heuristics are only essential in natural terrains where some pattern in obstacle density exists or in random terrains of high complexity.

Actual natural terrains, although usually best modeled by our natural terrains, could possibly be more similar to the random. Since the genetic algorithm produced search strategies are substantially better for our natural terrains and as least as good as standard search schemes for random terrains, they should be advantageous to use on any actual natural terrain. This is of course contingent on the physical agent's dependence on minimal steps and its computational speed. If it's computational speed is sufficient to avoid delays before each step and/or minimal steps are essential, the genetic algorithm produced scheme should always be used.

Appendix C shows a comparison of the average time required for each strategy to search from start to goal for each of the terrains. As expected, the more complicated strategies require additional computation time, but are not considered slow enough to prohibit their use except in cases of high speed agents with slow computational speed.

IX. CONCLUSIONS

Heuristics previously used for search of an unknown space by a physical agent are distance from goal and distance from current. These are insufficient to minimize energy expenditure (steps taken) when some general knowledge of the area is known. The additional heuristics found to be pertinent are distance from start, crowding factors which account for obstacle node density around the considered frontier node, move-away factor which encourages reduction of the search space, and momentum which avoids wasted steps in course variations. These seven heuristics with their proper individual biases were found to be superior to standard search schemes. In this thesis we showed that genetic algorithms can be effectively used to develop optimal heuristic biases that are adaptable to unknown search spaces if some general knowledge of the search space is available. Training done with randomly generated search spaces having common characteristics lead to robust search schemes which are, on the average, more fit than previously used strategies.

We believe that this methodology of identifying all possible heuristics, fitting them into a binary representation, and applying genetics-based training is also applicable to a multitude of real-time search/optimization problems. Tests in other specific areas are needed to prove our conjecture. In addition, further research could be done in the application of more advanced genetic algorithms. Our results showed significant improvement using only basic genetics-based concepts, advanced techniques should continue to improve the effectiveness of resultant strategies.

LIST OF REFERENCES

[De75] DeJong, Kenneth A., "An Analysis of the Behavior of a Class of Genetic Adaptive Systems." Doctoral Dissertation, Department of Computer and Communication Sciences, University of Michigan. 1975.

[De92] DeJong, Kenneth A., "Genetic Algorithms Are NOT Functional Optimizers." Computer Science Department, George Mason University. 1992.

[Go89] Goldberg, David E., Genetic Algorithms in Search, Optimization and Machine Learning. Reading, Ma: Addison-Wesley. 1989.

[Ha68] Hart, P.E., Nilsson, N.J., and Raphael B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Trans. Syst. Sci. Cybern., 4. 1968.

[Ko90] Korf, Richard E., "Real-Time Heuristic Search." Artificial Intelligence 42. 1990.

[Pa89] Papadimitrion, C.H. and Yannakakis, M., "Shortest Paths Without a Map," Proc. of the 1989 ICALP Conference. 1989.

[Sh91] Shing, Man-Tak and Mayer, Michael M., "Persistence Search - A New Search Strategy for the Dynamic Shortest Path Problem," Technical report NPSCS-91-011, Computer Science Dept., Naval Postgraduate School. 1991.

[St77] Stansfield, William D., The Science of Evolution. New York, NY: Macmillan. 1977.

[Wi92] Winston, Patrick H., Artificial Intelligence. Reading, Ma: Addison-Wesley. 1992.

APPENDIX A

TERRAIN DEVELOPMENT FROM A SAMPLE DENSITY MATRIX

Terrains are randomly produced using a density matrix as a guide. Figure 28 shows a density matrix that was used to develope the terrain is shown in figure 29. This density matrix was not used for our analysis, but helps to make clear the relationship between the density matrix and the actual terrain.

The density matrix is stored as a text file as shown in the figure. At each cycle for training or iteration for testing the density information is used to form a new terrain. Each hexadecimal number represents the desired density for a 4x4 area. The actual obstacle placement is random. Compare figures 28 and 29. The top left 4x4 area was filled in by checking if a random number (between 0 and 15) is less than 4 at each node. This should on the average happen 4 out of 16 times making the obstacle count of each 4x4 area equal 4. The top left 4x4 is the average case with 4 out of the 16 nodes being obstacles.

The remaining 4x4 areas are filled out in a similar fashion. The start and goal nodes are chosen at random in the (2,2) and (13,13) areas as also demonstrated in figure 29.

```
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 2 0 3 0 4 0 5 0 6 0 7 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
8 0 9 0 a 0 b 0 c 0 d 0 e 0 f 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 28 : Sample Density Matrix

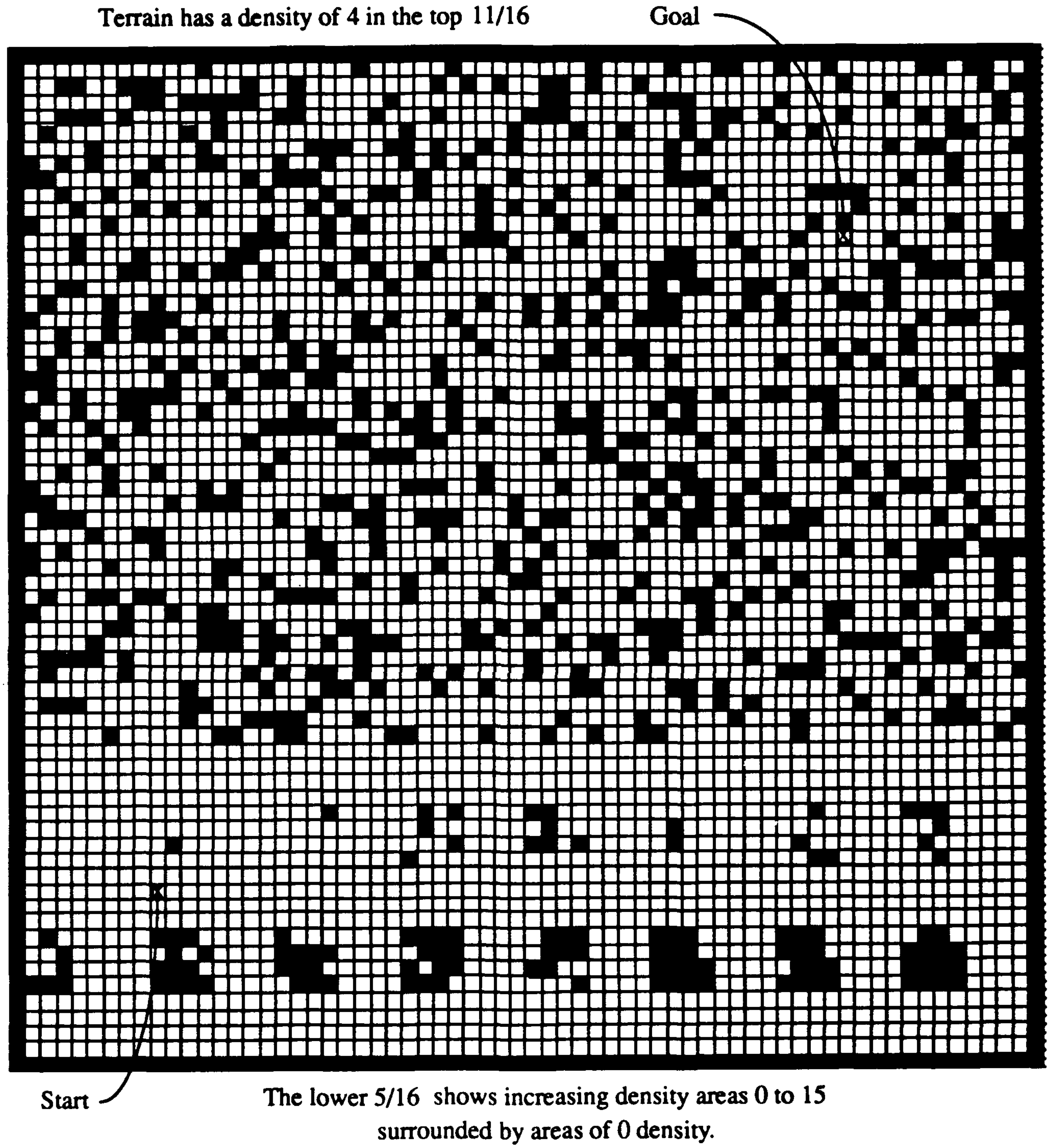

Figure 29 : One of Many Possible Resultant Terrains

APPENDIX B

RANDOM TERRAIN DENSITY MATICES

d	2	b	e	f	1	6	b	f	e	b	b	1	0	f	f
b	a	8	6	8	1	2	4	f	9	9	b	6	b	c	3
d	8	2	d	9	8	8	8	6	3	3	7	3	2	7	f
c	f	5	4	0	8	8	f	1	2	b	7	d	7	b	b
f	d	8	8	5	0	1	c	4	4	3	7	7	a	6	3
9	c	8	a	4	0	9	5	2	4	d	0	c	8	b	b
5	3	4	a	3	5	6	7	9	a	f	0	4	5	4	e
1	c	8	5	c	1	b	f	6	8	f	2	0	a	d	5
d	1	f	0	6	6	8	0	0	7	0	4	c	4	2	e
0	a	3	d	c	e	c	2	6	b	4	6	5	1	b	2
3	b	2	9	1	a	9	1	1	a	5	e	e	8	c	f
2	f	c	e	e	8	0	4	3	4	b	8	6	6	a	9
1	c	2	2	7	c	3	8	6	9	6	4	1	2	3	3
2	f	2	0	7	2	4	a	7	f	2	d	6	c	6	7
9	8	a	0	4	d	8	a	6	f	f	7	1	2	b	3
2	d	3	9	f	8	4	6	7	6	3	d	3	9	5	c

Figure 30 : Random Terrain One

6	4	6	c	d	c	8	9	a	d	a	c	4	6	e	5
4	f	7	b	5	9	0	4	5	5	a	2	8	d	9	e
1	0	b	f	c	3	8	6	0	2	2	4	8	0	a	d
f	1	8	4	a	9	8	f	e	3	2	6	0	b	5	2
b	0	1	7	3	9	e	4	c	0	8	4	1	2	1	0
3	a	5	e	3	d	d	1	0	f	8	1	b	d	3	6
d	4	e	0	d	c	4	9	c	d	e	d	f	f	e	3
9	3	1	c	0	e	e	1	e	6	2	9	3	5	f	0
9	d	0	6	9	5	0	6	2	e	3	1	d	1	4	7
4	5	3	5	4	1	6	2	7	8	b	a	d	a	a	6
8	b	c	1	0	c	7	2	a	b	3	8	c	8	f	1
d	2	6	1	4	c	3	b	4	e	6	1	9	0	7	1
b	3	2	b	0	a	d	a	5	1	2	1	9	1	2	6
4	8	8	8	4	b	3	8	a	9	9	3	a	0	4	5
4	6	1	4	0	e	e	5	f	1	7	8	2	9	f	6
2	7	e	6	2	2	f	c	b	8	f	5	9	3	b	d

Figure 31 : Random Terrain Two

2	c	a	4	5	0	0	1	e	1	e	8	8	e	2	d
0	b	b	7	1	9	8	4	d	5	6	6	0	5	5	2
1	0	7	7	0	7	8	e	8	6	6	0	4	8	e	5
3	9	c	4	2	5	8	f	a	f	6	a	4	b	d	6
b	4	d	b	b	5	a	4	c	0	4	0	9	2	5	c
b	2	1	e	7	9	d	1	8	3	c	d	f	9	3	a
d	0	6	8	5	0	c	1	0	1	2	9	3	7	6	f
9	7	d	0	0	a	2	9	e	e	6	d	7	9	7	4
9	d	c	e	d	9	0	e	a	2	7	d	9	d	c	3
4	9	3	5	4	5	e	2	3	4	f	a	d	6	e	6
4	b	4	1	4	4	f	e	6	7	b	0	4	8	3	9
1	6	e	5	c	c	7	f	0	6	a	d	d	8	3	1
3	7	2	7	c	2	5	2	9	1	2	d	9	5	6	a
c	4	0	8	0	7	7	0	e	1	d	b	a	0	c	d
8	e	5	4	0	a	6	9	b	9	7	4	e	d	f	a
2	f	2	2	6	a	3	4	b	0	f	5	1	b	3	9

Figure 32 : Random Terrain Three

b	e	5	2	3	b	2	f	9	0	d	9	b	4	1	3
9	0	a	c	e	1	6	4	3	1	7	d	2	7	2	d
5	8	0	9	3	2	8	c	2	5	5	d	9	6	1	3
6	b	f	4	c	6	8	f	7	0	d	9	7	f	7	d
7	7	6	a	9	e	7	c	4	c	9	d	3	a	0	9
5	0	e	2	6	6	1	d	6	e	7	e	e	e	b	5
5	1	0	e	f	7	a	3	3	4	1	6	e	1	0	4
1	e	6	7	4	7	5	b	6	c	9	4	a	4	9	f
5	9	d	4	0	8	8	4	c	9	a	a	a	a	e	c
8	4	3	d	c	8	8	2	4	1	6	e	5	f	d	a
9	b	e	9	3	6	d	f	f	8	9	a	2	8	6	b
c	9	8	8	2	0	a	6	1	0	5	6	0	2	0	9
d	e	2	0	5	0	f	4	8	9	e	a	1	4	5	d
e	d	6	0	d	0	6	e	1	b	4	1	e	4	a	b
3	c	c	8	c	b	c	4	4	b	f	5	f	4	3	d
2	9	d	f	9	4	e	a	f	2	b	d	7	5	[illegible]	a

Figure 33 : Random Terrain Four

APPENDIX C

SEARCH STRATEGY COMPUTATIONAL TIME

TABLE 1: AVERAGE SECONDS REQUIRED TO SEARCH EACH TERRAIN

	Best First	Persistence	Hill Climb	Real-Time-A*	GA-Produced
Central Mountain	0.0239	0.0274	0.0065	0.0163	0.0615
Single Left Ridge	0.0232	0.0264	0.0075	0.0176	0.0694
Singe Right Ridge	1.3051	0.1496	0.0199	0.1405	0.2718
Double Ridge	3.2553	1.3562	0.0420	0.4131	2.3420
Single Left Plateau	0.1071	0.0982	0.0167	0.0626	0.1481
Single Left Plateau With Ridges	0.0769	0.0856	0.0146	0.0487	0.1310
Random One	0.0177	0.0265	0.0064	0.0114	0.0478
Random Two	0.0313	0.0267	0.0056	0.0150	0.0644
Random Three	0.0294	0.0258	0.0065	0.0151	0.0521
Random Four	0.1629	0.2255	0.0165	0.0481	0.1563

APPENDIX D

PROGRAM C CODE

ga_search.h

```
/*
  File:        ga_search.h
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    This file holds all the header information needed for all
ga_search files.
*/

#include <stdio.h>

#define F 0
#define T 1

#define UNTOUCHED  0
#define OBSTACLE   1
#define VISITED    2
#define FRONTIER   3
#define START      4
#define GOAL       5
#define CURRENT    6
#define SHORTEST   7
#define X          8

#define NUM  0

#define N    1
#define E    2
#define S    3
#define W    4

#define NE   1
#define SE   2
#define SW   3
#define NW   4

#define MASK0 0     /* 0000 */
#define MASK1 8     /* 1000 */
#define MASK2 12    /* 1100 */
#define MASK3 14    /* 1110 */
#define MASK4 15    /* 1111 */
```

```
#define NA                  -1              /* not applicable; for frontier_in-
dex */
#define BIG_NUMBER          10000
#define STANDARD_DENSITY    4
#define PROB_BIT_MUTATE     50          /*  x/10000 prob of mutate */

#define SQRT2               1.414213562

#define rand16() ((random()/13) % 16)  /* return rand int from 0 to 15 */
#define rand8()  ((random()/13) % 8)   /* return rand int from 0 to 7 */
#define rand5()  ((random()/13) % 5)   /* return rand int from 0 to 4 */
#define rand10000() (random() % 10000) /* return rand int from 0 to 9999 */

/* Graphics definitions */
#define SLEEPTIME   2

#define ASTITLE     "A* Search                    "
#define GATITLE     "Genetic Algorithm Produced Search"
#define PERSTITLE   "Persistence Search           "
#define BFTITLE     "Best First Search            "
#define HCTITLE     "Hill Climb Search            "
#define RTASTITLE   "Real Time A* Search          "


/*  Node record for terrain */

struct node_rec
{
  int xi;
  int yi;
  float x;     /* for graphics */
  float y;     /* for graphics */
  int state;     /* UNTOUCHED, OBSTACLE, VISITED, or FRONTIER */
  int back_track_state;  /*  UNTOUCHED, VISITED, or OBSTACLE */
  float subtotal;        /* includes all but dist from current & move away */
  float dist_from_start;
  float dist_from_goal;
  float dist_from_current;
  struct node_rec *predecessor;   /* points to predecessor for a-star search */
  int frontier_index;              /* position in frontier heap */
  struct node_rec *qnext;          /* next in q for dist from current DFS */
  struct node_rec *qreset;         /* reset link list after back DFS */
```

```
/*  Individual record for population */

struct factor_struct {
  unsigned int place_holder : 4;
  unsigned int start_dist   : 4;
  unsigned int goal_dist    : 4;
  unsigned int current_dist : 4;
  unsigned int crowd_sides  : 4;
  unsigned int crowd_diag   : 4;
  unsigned int move_away    : 4;
  unsigned int momentum     : 4;
} ;

union chrom_union {
  struct factor_struct factor;
  unsigned int alleles;
} ;
struct individual_struct {
  union chrom_union chrom;
  int fitness;
  float fit_sum;
  int previous_index;
} ;


/* global variables */
extern int heap_size;

/*  Functions listed under file */

/* astar.c */
float a_star();
int update_astar_frontier();

/* rt_astar.c */
float rta_star();
int update_rtastar_adjacent();
int insert();

/* hill_climb.c */
float hill_climb();
struct node_rec *move_adjacent();
struct node_rec *find_best();

/* test.c */
int test();
```

```
/* train.c */
int train();

/* tsearch.c */
float search();

/* psearch.c */
float psearch();
int p_update_frontier_list();
struct node_rec *p_pick_best_frontier();

/* bfsearch.c */
float bfsearch();
int bf_update_frontier_list();
struct node_rec *bf_pick_best_frontier();

/* tsetup.c */
int get_seed();
unsigned int get_pers_chrom();
int read_density_file();
int make_array();
int make_node();
struct node_rec *find_node();

/* tpopulation */
int create_population();
struct individual_struct *new_individual();
int get_population();

/* tprint.c */
int print_density();
int print_node();
int print_population();

/* tfrontier.c */
int update_frontier_list();
struct node_rec *pick_best_frontier();
int update_adjacent_obstacles();
float update_dist();
int update_crowd_sides();
int update_crowd_diag();
int calc_move_away();
int calc_momentum();
float compute_subtotal();
```

```
/* theap.c */
int insert_heap();
int delete_heap();
int move_heap();
int swap();

/* tmisc.c */
float compute_shortest();
int equalf();
/* update_dist_start(); */
int gen_xi();
int gen_yi();
int show_least_nodes();

/* evolve.c */
int evolve();
int create_mate_heap();
int crossover();
int allele_crossover();
int bit_crossover();
int get_mask();

/* eheap.c */
int insert_mate_heap();
int pop_mate_heap();
int move_mate_heap();
int swap_num();

/*tdisplay.c */
int initialize();
int draw_terrain();
int show_mouse();
int draw_nodes();
int draw_grid();
void squaref();
void square();
```

main.c

```
/*
  File:        main.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    This is the control for user input and call of train or test.
Ten command live arguments are optional.  The syntax for a call is as follows
(the 0 argument is the program call):  (0) t;  (1) 0 for train, 1 for test;
(2) random_seed; (3) input population file name; (4) input terrain density file
name; (5) start region on X axis; (6) start region on Y axis; (7) goal region
on X axis; (8) goal region on Y axis; (9) number of generations if training,
iterations if testing; (10) cycles per generation if training, array position
of best individual in the GA_produced population for testing; (11) file name
for out population if training, hexadecimal representation of best Persistence
search scheme.
*/

#include "ga_search.h"


/* ***************** main ********************************************** */

main( argc, argv )
int argc;
char *argv[];
{
  struct individual_struct *individual[32];
  int arg_seed = 0;
  char arg_population[32];
  char arg_population_out[32];
  char arg_density[32];
  int choice = 0;
  int sx = 2;
  int sy = 2;
  int gx = 4;
  int gy = 2;
  int iterations = 25;
  int generations = 3;
  int cycles_per_generation = 2;
  int best_individual = -1;
  unsigned int pers_chrom;
  char hname[64];
  int hnlength;
```

```
gethostname( hname, hnlength );

strcpy( arg_population,     "                                        " );
strcpy( arg_population_out, "popx.out                                " );
strcpy( arg_density,        "                                        " );

switch ( argc )
  {
    case 12:
      sscanf( argv[11], "%s", arg_population_out );
      sscanf( argv[11], "%x", &pers_chrom );
    case 11:
      sscanf( argv[10], "%d", &cycles_per_generation );
      sscanf( argv[10], "%d", &best_individual );
    case 10:
      sscanf( argv[9], "%d", &generations );
      sscanf( argv[9], "%d", &iterations );
    case 9:
      sscanf( argv[8], "%d", &gy );
    case 8:
      sscanf( argv[7], "%d", &gx );
    case 7:
      sscanf( argv[6], "%d", &sy );
    case 6:
      sscanf( argv[5], "%d", &sx );
    case 5:
      sscanf( argv[4], "%s", arg_density );
    case 4:
      sscanf( argv[3], "%s", arg_population );
    case 3:
      sscanf( argv[2], "%d", &arg_seed );
```

```
        case 2:
          sscanf( argv[1], "%d", &choice );
          switch ( choice )
            {
            case 0:
              train(individual,arg_seed,arg_population,arg_density,sx,sy,gx,gy,
                    generations,cycles_per_generation,arg_population_out,hname );
              put_population( individual, arg_population_out );
              break;
            case 1:
              individual[0] = NULL;
              test(individual,arg_seed,arg_population,arg_density,sx,sy,gx,gy,
                    iterations,best_individual,pers_chrom);
              break;
            case 2:
              train(individual,arg_seed,arg_population,arg_density,sx,sy,gx,gy,
                    generations, cycles_per_generation,arg_population_out,hname);
              test(individual,arg_seed,arg_population,arg_density,sx,sy,gx,gy,
                    iterations,best_individual,pers_chrom);
              put_population( individual, arg_population_out );
              break;
            }
          break;
        case 1:
          train(individual,arg_seed,arg_population,arg_density,sx,sy,gx,gy,
                generations,cycles_per_generation,arg_population_out,hname);
          put_population( individual, arg_population_out );
        }
}
```

train.c

```
/*
  File:        train.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Called by main to train a population of 32 individuals.  If no
input population, a random one is generated.
*/

#include "ga_search.h"


/* ***************** train ******************************************** */

train( individual, arg_seed, arg_population, arg_density, sx, sy, gx, gy, gen-
erations, cycles_per_generation, arg_population_out, hname )
struct individual_struct *individual[32];
int arg_seed;
char arg_population[32];
char arg_density[32];
int sx, sy, gx, gy;
int generations, cycles_per_generation;
char arg_population_out[32];
char hname[64];
{
    int density[16][16];
    struct node_rec *node[66][66];
    int gen, cycle, i, rs, short_count;
    float shortest_path;
    int dummy;

    rs = arg_seed ? arg_seed : get_seed();
    srandom(rs);  /* seed the random generator */

    printf("\nRandom seed is %d", rs );

    get_population( individual, arg_population );

    read_density_file( density, arg_density );
```

```
for( gen = 1; gen <= generations; gen++ ) {
  printf("\n gen = %d   (cycle,rs) = ", gen);
  for( cycle = 1; cycle <= cycles_per_generation; cycle++ ) {
    rs = rs + 1;
    short_count = 0;
    while( (shortest_path = a_star(sx,sy,gx,gy,rs,density,node ))
            > ( BIG_NUMBER - 1.0 ) ) {
       if( short_count > 1000000000 ) {
         printf("\nPROGRAM ABORTED - iteration %d - no shortest path\n",i);
         return(F);
       }
       else
         rs = rs + 1;
    }
    printf(" (%d,%d)", cycle, rs );
    for( i=0; i<32; i++ ) {
      if( cycle == 1 )
        individual[i]->fit_sum = shortest_path / search( sx,sy,gx,gy,
                 individual[i]->chrom.factor,rs,density,node,&dummy );
      else
        individual[i]->fit_sum = individual[i]->fit_sum + shortest_path /
              search( sx,sy,gx,gy,individual[i]->chrom.factor,rs,density,
              node,&dummy );
      if( cycle == cycles_per_generation )
        individual[i]->fitness = (int)((individual[i]->fit_sum /
              cycles_per_generation) * 100.0);
    }
  }
  if (gen == generations - 1 )
    cycles_per_generation = cycles_per_generation + 10;

  evolve( individual, rs );
  put_rs( rs );
  if ( ( (gen % 10) == 0 ) || (gen < 10) )
    put_gen(gen,arg_population_out,hname,rs);
             /* put gen to a standard update file */
  if ( (gen % 50) == 0 ) {
    put_population( individual, arg_population_out );
  }
}
}
```

```
/* ***************** put_gen ****************************************** */
/* Called by train to continually store status information to a file in the
directory of execution */

put_gen( gen, arg_population_out, hname, rs )
int gen;
char arg_population_out[32];
char hname[64];
int rs;
{
  FILE *gen_file, *fopen();

  gen_file = fopen("running.update", "a");

  fprintf(gen_file," %s %s gen = %d  rs = %d\n", hname, arg_population_out,
gen, rs);

  fclose(gen_file);
}


/* ***************** put_rs ****************************************** */
/* Puts random seed info to a file in the directory of execution */

put_rs( rs )
int rs;
{
  FILE *rs_file, *fopen();

  rs_file = fopen("rs.update", "a");

  fprintf(rs_file," %d ", rs);

  fclose(rs_file);
}
```

test.c

```
/*
  File:        test.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Called by main to perform a comparative test of search
strategies.  The default is for all individuals of the population to be
tested, unless a specific individual is specified.
*/

#include "ga_search.h"
#include <sys/time.h>


/* ***************** test  ******************************************** */

test( individual, arg_seed, arg_population, arg_density, sx, sy, gx, gy,
      iterations, best_individual, pers_chrom )
struct individual_struct *individual[32];
int arg_seed;
char arg_population[32];
char arg_density[32];
int sx, sy, gx, gy;
int iterations;
int best_individual;
unsigned int pers_chrom;
{

    struct individual_struct *best_first;
    struct individual_struct *persistence_search;
    int density[16][16];
    struct node_rec *node[66][66];

    int i, rs, k, short_count;

    float shortest_path;
    float realtime_astar_fit_sum;
    int realtime_astar_fitness;
    float hill_climb_fit_sum;
    int hill_climb_fitness;
```

```
float temp;
float ga_t = 0.0;
float ga_ticks = 0.0;
fl at bf_t = 0.0;
float bf_ticks = 0.0;
float pers_t = 0.0;
float pers_ticks = 0.0;
float hc_t = 0.0;
float hc_ticks = 0.0;
float rta_t = 0.0;
float rta_ticks= 0.0;

long sec, usec;
struct timeval *tvp = (struct timeval *)malloc(sizeof(struct timeval));
struct timezone *tzp = (struct timezone *)malloc(sizeof(struct timezone));

best_first =
     (struct individual_struct *)malloc(sizeof(struct individual_struct));
persistence_search =
     (struct individual_struct *)malloc(sizeof(struct individual_struct));
best_first->chrom.alleles     = 0xe0100000;
persistence_search->chrom.alleles = pers_chrom;
best_first->fit_sum     = 0.0;
persistence_search->fit_sum = 0.0;

rs = arg_seed ? arg_seed : get_seed();
srandom(rs);  /* seed the random generator */

printf("\nRandom seed is %d", rs);
if ( individual[0] == NULL )
  get_population( individual, arg_population );

for(k=0;k<32;k++)
  individual[k]->fit_sum = 0.0;

read_density_file( density, arg_density );

printf("\n(iteration,rs) ");
```

```
for( i = 1; i <= iterations; i++ ) {
  rs = rs + 1;
  short_count = 0;
  while ( (shortest_path = a_star( sx, sy, gx, gy, rs, density, node ))
          > ( BIG_NUMBER - 1.0 ) ) {
    if( short_count > 1000000000 ) {
      printf("\n PROGRAM ABORTED - iteration %d - no shortest path\n", i );
      return(F);
    }
    else {
      printf(" (%d,%d)",i,rs );
      rs = rs + 1;
    }
  }
  printf(" (%d,%d)",i,rs );
  if (best_individual == -1)
    for(k=0;k<32;k++) {
      temp = search( sx,sy,gx,gy,individual[k]->chrom.factor,rs,density,
                     node,&ga_t );
      individual[k]->fit_sum =
                     individual[k]->fit_sum + shortest_path / temp;
      if (temp < shortest_path )
        printf("\nSHORTEST PATH > ACTUAL PATH ");
    }
  else {
    k = best_individual;
    temp = search( sx,sy,gx,gy,individual[k]->chrom.factor,rs,density,
                   node,&ga_t );
    individual[k]->fit_sum = individual[k]->fit_sum + shortest_path / temp;
    if (temp < shortest_path )
      printf("\nSHORTEST PATH > ACTUAL PATH ");
    ga_ticks = ga_ticks + ga_t;
    best_first->fit_sum = best_first->fit_sum + shortest_path /
                          bfsearch( sx,sy,gx,gy,best_first->chrom.factor,
                                    rs,density,node,&bf_t );
    bf_ticks = bf_ticks + bf_t;
    persistence_search->fit_sum = persistence_search->fit_sum +
               shortest_path / psearch( sx,sy,gx,gy,
               persistence_search->chrom.factor, rs,density,node,&pers_t );
    pers_ticks = pers_ticks + pers_t;
    hill_climb_fit_sum = hill_climb_fit_sum + shortest_path /
                   hill_climb( sx, sy, gx, gy, rs, density, node, &hc_t );
    hc_ticks = hc_ticks + hc_t;
    realtime_astar_fit_sum = realtime_astar_fit_sum + shortest_path /
                     rta_star( sx, sy, gx, gy, rs, density, node, &rta_t );
    rta_ticks = rta_ticks + rta_t;
  }
}
```

```
  if (best_individual == -1) /* no best individual input */
    for(k=0;k<32;k++)
      individual[k]->fitness =
              (int)((individual[k]->fit_sum / iterations) * 100.0);
  else
    individual[k]->fitness =
                        (int)((individual[k]->fit_sum / iterations) * 100.0);

  best_first->fitness = (int)((best_first->fit_sum / iterations) * 100.0);
  persistence_search->fitness =
              (int)((persistence_search->fit_sum / iterations) * 100.0);
  realtime_astar_fitness =
              (int)((realtime_astar_fit_sum / iterations) * 100.0);
 hill_climb_fitness = (int)((hill_climb_fit_sum / iterations) * 100.0);

  if (best_individual == -1)
    print_population( individual );
  else {
    printf("\n ga-produced %2d %7.3f %7.3f   %x", individual[k]->fitness,
            individual[k]->fit_sum, ga_ticks, individual[k]->chrom.alleles);
    printf("\n best_first  %2d %7.3f %7.3f   %x", best_first->fitness,
            best_first->fit_sum, bf_ticks, best_first->chrom.alleles);
    printf("\n persistence %2d %7.3f %7.3f   %x",
            persistence_search->fitness, persistence_search->fit_sum,
            pers_ticks, persistence_search->chrom.alleles);
    printf("\n hill climb  %2d %7.3f %7.3f", hill_climb_fitness,
            hill_climb_fit_sum, hc_ticks );
    printf("\n RTA star    %2d %7.3f %7.3f", realtime_astar_fitness,
            realtime_astar_fit_sum, rta_ticks );
  }
  printf("\n");
}
```

tsetup.c

```
/*
  File:         tsetup.c
  Programmer:   g.b. parker
  Environment:  any
  Language:     C
  Date:          9 july 92
  Revised:
  Comments:     Setup functions
*/

#include "ga_search.h"


/* ***************** get_seed ***************************************** */
/* Interfaces with user to get random seed */

get_seed()
{
    char nl[1];   /* absorbs new_line after seed entry */
    int rand_seed;
    printf("\nEnter random seed or 0 (system assign seed): ");
    scanf("%d",&rand_seed);
    gets(nl);
    if (rand_seed != 0)
       return rand_seed;
    else
       return getpid();
}


/* ***************** get_pers_chrom *********************************** */
/* Interfaces with user to get Persistence chromosome */

unsigned int get_pers_chrom()
{
    char nl[1];   /* absorbs new_line after seed entry */
    unsigned int pers_chrom;
    printf("\nEnter Persistence chromosome (8 hex digits) or 0 (e0110000): ");
    scanf("%x",&pers_chrom);
    gets(nl);
    if (pers_chrom != 0)
       return( pers_chrom );
    else
       return( 0xe0110000 );
}
```

```
/* ***************** read_density_file ****************************** */
/* Reads density file from execution directory */

read_density_file( density, file_name )
int density[16][16];
char file_name[32];
{

  int i, j;
  FILE *density_file, *fopen();
  int not_end = T;
  int node_density = STANDARD_DENSITY;

  if( file_name[0] == ' ') {
    printf("\nEnter density file name: ");
    gets(file_name);
  }

  if ( (density_file = fopen(file_name,"r")) == NULL ) {
    printf("\nThe file does not exist, standard densities being used.");
    not_end = F;
  }
  for ( j=15 ; j>=0 ; j-- ) {
    for ( i=0 ; i<=15 ; i++ ) {
      if ( not_end && ( fscanf(density_file,"%x",&node_density) != EOF ) )
        density[i][j] = node_density;
      else {
        density[i][j] = STANDARD_DENSITY;
        not_end = F;
      }
    }
  }
  if(density_file != NULL)
    fclose(density_file);
}
```

```
/* ***************** make_array ************************************** */
/* Creates the node array on initial use, then resets records after that */

make_array( density, node )
int density[16][16];
struct node_rec *node[66][66];
{
     static int first = T;  /* indicates if first time to make array */

     int i, j;

     for(i=0;i<=65;i++) {
        make_node( node, i, 0, first );
        make_node( node, i, 65, first );
        node[i][0]->state = OBSTACLE;
        node[i][65]->state = OBSTACLE;
      }

     for(j=1;j<=64;j++) {
        make_node( node, 0, j, first );
        make_node( node, 65, j, first );
        node[0][j]->state = OBSTACLE;
        node[65][j]->state = OBSTACLE;
      }

     for(j=1;j<=64;j++)
        for(i=1;i<=64;i++) {
           make_node( node, i, j, first );
           if (rand16() < density[(i-1)/4][(j-1)/4])
              node[i][j]->state = OBSTACLE;
    }
  first = F;
}
```

```
/* ***************** make_node ************************************** */
/* Resets single node information */

make_node( node, xi, yi, first )
struct node_rec *node[66][66];
int xi;
int yi;
int first;  /* T of F */
{
  int k;

  if( first )  {
    node[xi][yi] = (struct node_rec *)malloc(sizeof(struct node_rec));
    node[xi][yi]->xi = xi;
    node[xi][yi]->yi = yi;
    node[xi][yi]->x = (float)xi;
    node[xi][yi]->y = (float)yi;
  }
  node[xi][yi]->state = UNTOUCHED;    /* OBSTACLE, VISITED, or FRONTIER */
  node[xi][yi]->back_track_state = UNTOUCHED;    /* VISITED, or FRONTIER */
  node[xi][yi]->subtotal = BIG_NUMBER;
  node[xi][yi]->dist_from_start = BIG_NUMBER;
  node[xi][yi]->dist_from_goal = BIG_NUMBER;
  node[xi][yi]->dist_from_current = 0.0;
  node[xi][yi]->predecessor = node[xi][yi];
                          /* points to predecessor for a-star search */
  node[xi][yi]->frontier_index = NA;   /* not have index to frontier_heap */
  node[xi][yi]->qnext = NULL;
                          /* points to next in q for dist from current DFS */
  node[xi][yi]->qreset = NULL;   /* reset link list after back DFS */
}
```

```
/* ****************** find_node ***************************************** */
/* Picks a random node in the designated density area.  Used to identify
start and goal nodes. */

struct node_rec *find_node( node, dens_col, dens_row)
struct node_rec *node[66][66];
int dens_col;
int dens_row;
{
  int k, xi, yi, base_x, base_y;

  base_x = (dens_col * 4) + 1;
  base_y = (dens_row * 4) + 1;

  for (k=0;k<100;k++) {
    xi = base_x + (rand16() % 4);
    yi = base_y + (rand16() % 4);

    if (node[xi][yi]->state == UNTOUCHED)
      return node[xi][yi];
  }
  return node[xi][yi];
}
```

tpopulation.c

```
/*
  File:        tpopulation
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Functions dealing with population creation/storage
*/

#include "ga_search.h"


/* ***************** create_population ****************************** */
/* Generates a population of random individuals */

create_population( individual )
struct individual_struct *individual[32];
{
  int k;

  for (k=0;k<32;k++) {
    individual[k] =
         (struct individual_struct *)malloc(sizeof(struct individual_struct));
    new_individual( individual[k] );
  }
}


/* ***************** new_individual ********************************** */
/* Sets initial values of individual records fields */

struct individual_struct *new_individual( ind )
struct individual_struct *ind;
{
  ind->chrom.factor.place_holder = 0xf;
  ind->chrom.factor.start_dist   = rand16();
  ind->chrom.factor.goal_dist    = rand16();
  ind->chrom.factor.current_dist = rand16();
  ind->chrom.factor.crowd_sides  = rand16();
  ind->chrom.factor.crowd_diag   = rand16();
  ind->chrom.factor.move_away    = rand16();
  ind->chrom.factor.momentum     = rand16();
  ind->fitness      = 0;
  ind->fit_sum      = 0.0;
  ind->previous_index = 99;
}
```

```
/* ***************** get_population ********************************** */
/* Reads population from a file */

get_population( individual, file_name )
struct individual_struct *individual[32];
char file_name[32];
{
  int i;
  FILE *population_file, *fopen();
  int not_end = T;
  unsigned int alleles;

  if ( file_name[0] == ' ' ) {
    printf("\nEnter population file name: ");
    gets(file_name);
  }

  if ( (population_file = fopen(file_name,"r")) == NULL ) {
    printf("\nThe file does not exist, random population being used.");
    create_population( individual );
  }
  else {

    for ( i=0 ; i<32 ; i++ ) {
      individual[i] =
          (struct individual_struct *)malloc(sizeof(struct individual_struct));
      individual[i]->fitness        = 0;
      individual[i]->fit_sum        = 0.0;
      individual[i]->previous_index = 99;

      if ( not_end && ( fscanf(population_file,"%x",&alleles) != EOF ) )
        individual[i]->chrom.alleles = alleles;

      else {
        new_individual( individual[i] );
        not_end = F;
      }
    }
    fclose(population_file);
  }
}
```

```
/* ***************** put_population ********************************** */
/* Puts the population to a designated file */

put_population( individual, file_name )
struct individual_struct *individual[32];
char file_name[32];
{
  int i;
  FILE *population_file, *fopen();

  if( file_name[0] == ' ' ) {
    printf("\nEnter output population file name: ");
    gets(file_name);
  }

  population_file = fopen(file_name,"w");

  for ( i=0 ; i<32 ; i++ )
    fprintf(population_file,"%x\n",individual[i]->chrom.alleles);

  fclose(population_file);
}
```

astar.c

```
/*
  File:        astar.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    A-star search - Finds the shortest path
*/

#include "ga_search.h"

int heap_size;


/* ***************** a_star ******************************************* */
float a_star( sx, sy, gx, gy, random_seed, density, node )
int sx,sy,gx,gy;    /* position in density array for start & goal */
int random_seed;
int density[16][16];
struct node_rec *node[66][66];
{
    struct node_rec *current, *start, *goal;
    struct node_rec *frontier_heap[4096];
    int k;

#ifdef IRIS
    union chrom_union dummy_cu;
    dummy_cu.alleles = 0;
#endif

    heap_size = 0;
    srandom(random_seed);  /* seed the random generator */
    make_array(density, node);
    start = find_node(node, sx, sy);
    start->state = START;
    start->dist_from_start = 0.0;
    goal = find_node(node, gx, gy);
    goal->state = GOAL;
    for (k=0;k<4096;k++)
      frontier_heap[k] = NULL;
    current = start;
```

```
#ifdef IRIS
    /* initialize the IRIS system */
    initialize(ASTITLE);
    draw_terrain(node,start,goal,current,goal->dist_from_start,
                 dummy_cu.factor);
#endif

    while ( current != goal) {
      update_astar_frontier( node,current,frontier_heap,start,goal );
      if( heap_size == 0) break;
      current = frontier_heap[0];
      delete_heap( frontier_heap, frontier_heap[0] );
    }

#ifdef IRIS
    draw_terrain(node,start,goal,current,goal->dist_from_start,
                 dummy_cu.factor);
    sleep( SLEEPTIME );
#endif

    goal->state = GOAL;
    start->state = START;

    return( goal->dist_from_start );
}
```

```
/* ***************** update_astar_frontier ************************** */
update_astar_frontier( node, c, frontier_heap, start, goal )
struct node_rec *node[66][66];
struct node_rec *c;          /* current */
struct node_rec *frontier_heap[4096];
struct node_rec *start;
struct node_rec *goal;
{
  int xi, yi, base_xi, base_yi, top_xi, top_yi;

  base_xi = c->xi ==  1 ?  1 : c->xi - 1;
  base_yi = c->yi ==  1 ?  1 : c->yi - 1;
  top_xi  = c->xi == 64 ? 64 : c->xi + 1;
  top_yi  = c->yi == 64 ? 64 : c->yi + 1;

  for (xi=base_xi;xi<=top_xi;xi++)
    for (yi=base_yi;yi<=top_yi;yi++)
      if ( (node[xi][yi]->state == UNTOUCHED) ||
           (node[xi][yi]->state == GOAL) ) {
        node[xi][yi]->state = FRONTIER;
        node[xi][yi]->predecessor = c;
        node[xi][yi]->dist_from_goal  = update_dist(node[xi][yi],goal);
        node[xi][yi]->dist_from_start =
                   c->dist_from_start + update_dist(node[xi][yi],c);
        node[xi][yi]->subtotal = node[xi][yi]->dist_from_goal +
                                 node[xi][yi]->dist_from_start;
        insert_heap( frontier_heap, node[xi][yi] );
      }

      else if ( node[xi][yi]->state == FRONTIER ) {
        if ( node[xi][yi]->dist_from_start >
             (c->dist_from_start + update_dist(node[xi][yi],c)) ) {
          node[xi][yi]->dist_from_start =
                       c->dist_from_start + update_dist(node[xi][yi],c);
          node[xi][yi]->subtotal = node[xi][yi]->dist_from_goal +
                                   node[xi][yi]->dist_from_start;
          move_heap( frontier_heap, node[xi][yi]->frontier_index );
        }
      }
}
```

hill_climb.c

```
/*
  File:        hill_climb.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Hill climb search - Finds best adjacent frontier node or back-
tracks
*/

#include "ga_search.h"
#include <sys/time.h>


/* ***************** hill_climb ************************************** */
float hill_climb( sx, sy, gx, gy, random_seed, density, node, ticks )
int sx,sy,gx,gy;    /* position in density array for start & goal */
int random_seed;
int density[16][16];
struct node_rec *node[66][66];
float *ticks;
{
    struct node_rec *current, *next, *start, *goal;
    int k = 1;

    long sec, usec;
    static struct timeval *tvp;
    static struct timezone *tzp;
    static int first = T;

#ifdef IRIS
    union chrom_union dummy_cu;
    dummy_cu.alleles = 0;
#endif

    if( first ) {
      tvp = (struct timeval *)malloc(sizeof(struct timeval));
      tzp = (struct timezone *)malloc(sizeof(struct timezone));
      first = F;
    }

    srandom(random_seed);  /* seed the random generator */
    make_array(density, node);
    start = find_node(node, sx, sy);
    start->state = START;
    start->dist_from_start = 0.0;
```

```
    goal = find_node(node, gx, gy);
    goal->state = GOAL;
    current = start;
    current->state = CURRENT;

#ifdef IRIS
    /* initialize the IRIS system */
    initialize(HCTITLE);
#endif

    gettimeofday(tvp, tzp);
    sec = tvp->tv_sec;
    usec = tvp->tv_usec;

    while ( current != goal) {
      next = move_adjacent( node, current, start, goal );

      if ( next != NULL )
        next->predecessor = current;
      else if ( current->predecessor != NULL )
        next = current->predecessor;
      else
        printf("\nNO SOLUTION - hill climb search");

      next->dist_from_start =
                     current->dist_from_start + update_dist( current, next );
      current->state = VISITED;
      current = next;
      current->state = CURRENT;
#ifdef IRIS
      draw_terrain(node,start,goal,current,next->dist_from_start,
                   dummy_cu.factor);
#endif
    }

    gettimeofday(tvp, tzp);
    *ticks = (float)(tvp->tv_sec - sec) + (tvp->tv_usec - usec)/1000000.0;

#ifdef IRIS
    sleep( SLEEPTIME );
#endif

    goal->state = GOAL;
    start->state = START;

    return( goal->dist_from_start );
}
```

```
/* ***************** move_adjacent ********************************** */
struct node_rec *move_adjacent( node, c, start, goal )
struct node_rec *node[66][66];
struct node_rec *c;         /* current */
struct node_rec *start;
struct node_rec *goal;
{
  struct node_rec *best;
  int xi, yi, base_xi, base_yi, top_xi, top_yi;

  best = NULL;
  base_xi = c->xi ==  1 ?  1 : c->xi - 1;
  base_yi = c->yi ==  1 ?  1 : c->yi - 1;
  top_xi  = c->xi == 64 ? 64 : c->xi + 1;
  top_yi  = c->yi == 64 ? 64 : c->yi + 1;

  for (xi=base_xi;xi<=top_xi;xi++)
    for (yi=base_yi;yi<=top_yi;yi++)
      switch ( node[xi][yi]->state )
        {
        case UNTOUCHED:
        case GOAL:
          node[xi][yi]->state = FRONTIER;
          node[xi][yi]->dist_from_goal  = update_dist( node[xi][yi], goal  );
          best = find_best( best, node[xi][yi] );
          break;
        case FRONTIER:
          best = find_best( best, node[xi][yi] );
          break;
        }
  return( best );
}


/* ***************** find_best ****************************************** */
/* Assigns the input node to best if appropriate */

struct node_rec *find_best( best, n )
struct node_rec *best;
struct node_rec *n;
{
  if ( best == NULL )
    best = n;
  else if ( n->dist_from_goal < best->dist_from_goal )
    best = n;
  return( best );
}
```

rt_astar.c

```
/*
  File:         rt_astar.c
  Programmer:   g.b. parker
  Environment:  any
  Language:     C
  Date:         20 feb 92
  Revised:       2 apr 92
  Comments:     RTA-star search - Finds best adjacent node visited or frontier
*/

#include "ga_search.h"
#include <sys/time.h>

int heap_size;


/* ***************** rta_star ***************************************** */

float rta_star( sx, sy, gx, gy, random_seed, density, node, ticks )
int sx,sy,gx,gy;     /* position in density array for start & goal */
int random_seed;
int density[16][16];
struct node_rec *node[66][66];
float *ticks;
{
    struct node_rec *current, *start, *goal;
    struct node_rec *best_two[2];
    int k = 1;

    long sec, usec;
    static struct timeval *tvp;
    static struct timezone *tzp;
    static int first = T;
#ifdef IRIS
    union chrom_union dummy_cu;
    dummy_cu.alleles = 0;
#endif

    if( first ) {
      tvp = (struct timeval *)malloc(sizeof(struct timeval));
      tzp = (struct timezone *)malloc(sizeof(struct timezone));
      first = F;
    }

    heap_size = 0;
    srandom(random_seed);  /* seed the random generator */
    make_array(density, node);
```

```
    start = find_node(node, sx, sy);
    start->state = START;
    start->dist_from_start = 0.0;
    goal = find_node(node, gx, gy);
    goal->state = GOAL;
    current = start;
    current->state = CURRENT;

#ifdef IRIS
    /* initialize the IRIS system */
    initialize(RTASTITLE);
#endif

    gettimeofday(tvp, tzp);
    sec = tvp->tv_sec;
    usec = tvp->tv_usec;

    while ( current != goal) {
      best_two[0] = NULL;
      best_two[1] = NULL;
      update_rtastar_adjacent( node, current, best_two, start, goal );
      if( best_two[0] == NULL ) break;
      if( best_two[1] == NULL )
        current->dist_from_goal = BIG_NUMBER;
      else
        current->dist_from_goal = best_two[1]->subtotal;
      current->state = VISITED;
      best_two[0]->dist_from_start =
                   current->dist_from_start + update_dist(current,best_two[0]);
      current = best_two[0];
      current->state = CURRENT;

#ifdef IRIS
      draw_terrain(node,start,goal,current,best_two[0]->dist_from_start,
                   dummy_cu.factor);
#endif
    }

    gettimeofday(tvp, tzp);
    *ticks = (float)(tvp->tv_sec - sec) + (tvp->tv_usec - usec)/1000000.0;

#ifdef IRIS
    sleep( SLEEPTIME );
#endif

    goal->state = GOAL;
    start->state = START;
    return( goal->dist_from_start );
}
```

```
/* ***************** update_rtastar_adjacent ************************ */

update_rtastar_adjacent( node, c, best_two, start, goal )
struct node_rec *node[66][66];
struct node_rec *c;         /* current */
struct node_rec *best_two[2];
struct node_rec *start;
struct node_rec *goal;
{
  int xi, yi, base_xi, base_yi, top_xi, top_yi;

  base_xi = c->xi ==  1 ?  1 : c->xi - 1;
  base_yi = c->yi ==  1 ?  1 : c->yi - 1;
  top_xi  = c->xi == 64 ? 64 : c->xi + 1;
  top_yi  = c->yi == 64 ? 64 : c->yi + 1;

  for (xi=base_xi;xi<=top_xi;xi++)
    for (yi=base_yi;yi<=top_yi;yi++)
      switch ( node[xi][yi]->state )
        {
        case UNTOUCHED:
        case GOAL:
          node[xi][yi]->state = FRONTIER;
          node[xi][yi]->predecessor = c;
          node[xi][yi]->dist_from_goal  = update_dist( node[xi][yi], goal  );
          node[xi][yi]->subtotal =
                 node[xi][yi]->dist_from_goal + update_dist( node[xi][yi], c );
          insert( best_two, node[xi][yi] );
          break;
        case FRONTIER:
        case VISITED:
          node[xi][yi]->subtotal =
                 node[xi][yi]->dist_from_goal + update_dist( node[xi][yi], c );
          insert( best_two, node[xi][yi] );
          break;
        case CURRENT:
          break;
        }
}
```

```
/* ****************** insert ******************************************** */
/* Assigns the input node to best or second_best as appropriate */

insert( best_two, n )
struct node_rec *best_two[2];
struct node_rec *n;
{
  if ( best_two[0] == NULL )
    best_two[0] = n;
  else if ( n->subtotal < best_two[0]->subtotal ) {
    best_two[1] = best_two[0];
    best_two[0] = n;
  }
  else if ( best_two[1] == NULL )
    best_two[1] = n;
  else if ( n->subtotal < best_two[1]->subtotal )
    best_two[1] = n;
}
```

bfsearch.c

```
/*
  File:        bfsearch.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Best First Search - modified standard bfs for use with real-
time search
*/

#include "ga_search.h"
#include <sys/time.h>


/* ***************** bfsearch ***************************************** */

float bfsearch( sx, sy, gx, gy, ind_chrom_factor, random_seed, density, node,
ticks )
int sx,sy,gx,gy;    /* position in density array for start & goal */
struct factor_struct ind_chrom_factor;
int random_seed;
int density[16][16];
struct node_rec *node[66][66];
float *ticks;
{
    struct node_rec *current, *previous, *start, *goal;
    struct node_rec *frontier_heap[4096];

    long sec, usec;
    static struct timeval *tvp;
    static struct timezone *tzp;
    static int first = T;

    float dist_traveled = 0.0;
    int k;

#ifdef IRIS
    union chrom_union dummy_cu;
    dummy_cu.alleles = 0;
#endif

    if( first ) {
      tvp = (struct timeval *)malloc(sizeof(struct timeval));
      tzp = (struct timezone *)malloc(sizeof(struct timezone));
      first = F;
    }
```

```
    srandom(random_seed);  /*  seed the random generator */
    heap_size = 0;
    make_array(density, node);

    start = find_node(node, sx, sy);
    start->state = START;
    start->dist_from_start = 0.0;
    goal = find_node(node, gx, gy);
    goal->state = GOAL;

    for (k=0;k<4096;k++)
      frontier_heap[k] = NULL;
    current = start;
    previous = start;
#ifdef IRIS
    /* initialize the IRIS system */
    initialize(BFTITLE);
#endif
    gettimeofday(tvp, tzp);
    sec = tvp->tv_sec;
    usec = tvp->tv_usec;

    while ( current != goal) {
      bf_update_frontier_list(node,current,previous,frontier_heap,start,goal);
      if( heap_size == 0) {
        printf("\nENDING SEARCH BEFORE GOAL - no more frontier");
        break;
      }
      previous = current;
      current = bf_pick_best_frontier(node,current,frontier_heap,goal);
      dist_traveled = dist_traveled + current->dist_from_current;
#ifdef IRIS
      draw_terrain(node,start,goal,current,dist_traveled,dummy_cu.factor);
#endif
    }

    gettimeofday(tvp, tzp);
    *ticks = (float)(tvp->tv_sec - sec) + (tvp->tv_usec - usec)/1000000.0;

#ifdef IRIS
    sleep( SLEEPTIME );
#endif

    goal->state = GOAL;
    start->state = START;

    return( dist_traveled );
}
```

```
/* ***************** bf_update_frontier_list ************************ */

bf_update_frontier_list( node, c, p, frontier_heap, start, goal )
struct node_rec *node[66][66];
struct node_rec *c;         /* current */
struct node_rec *p;         /* previous */
struct node_rec *frontier_heap[4096];
struct node_rec *start;
struct node_rec *goal;
{
  int xi, yi, base_xi, base_yi, top_xi, top_yi;
  float old_subtotal;

  base_xi = c->xi - 1;
  base_yi = c->yi - 1;
  top_xi  = c->xi + 1;
  top_yi  = c->yi + 1;

  for (xi=base_xi;xi<=top_xi;xi++)
    for (yi=base_yi;yi<=top_yi;yi++)
      if ( (node[xi][yi]->state == UNTOUCHED) || (node[xi][yi]->state == GOAL)
) {
        node[xi][yi]->state = FRONTIER;
        node[xi][yi]->predecessor = c;
        node[xi][yi]->dist_from_goal  = update_dist( node[xi][yi], goal );
        node[xi][yi]->subtotal = node[xi][yi]->dist_from_goal;
        insert_heap( frontier_heap, node[xi][yi] );
      }
}


/* ***************** bf_pick_best_frontier ************************* */
/* Finds best frontier node, returns it */

struct node_rec *bf_pick_best_frontier( node, current, frontier_heap, goal )
struct node_rec *node[66][66];
struct node_rec *current;
struct node_rec *frontier_heap[4096];
struct node_rec *goal;
{
  struct node_rec *best_ptr, *q, *qend, *qreset;
  float node_cost = BIG_NUMBER;
  float norm = 1.0; /* (current->dist_from_goal / 16.0);   normalize factor */
  int xi, yi, k;
  float steps;
  int done = F;
```

```
  best_ptr = frontier_heap[0];
  q = current;
  qend = current;
  qreset = current;
  current->back_track_state = VISITED;
  current->dist_from_current = 0.0;

  while ( !done && (q != NULL) ) {

    steps = q->dist_from_current + 1.0;
    for(k=0;k<8;k++) {
      if (k==4)
        steps =  q->dist_from_current + SQRT2;
      xi = gen_xi( k, q->xi );
      yi = gen_yi( k, q->yi );

      if(  ( (node[xi][yi]->state == VISITED) ||
             (node[xi][yi]->state == FRONTIER) ||
             (node[xi][yi]->state == START) ) &&
           node[xi][yi]->back_track_state == UNTOUCHED ) {
        node[xi][yi]->dist_from_current = steps;
        node[xi][yi]->back_track_state = VISITED;
        node[xi][yi]->qreset = qreset;
        qreset = node[xi][yi];
        if ( (node[xi][yi]->state == VISITED) ||
             (node[xi][yi]->state == START) ) {
          qend->qnext = node[xi][yi];
          qend = node[xi][yi];
        }
        else {    /* node[xi][yi]->state == FRONTIER */
          if ( node[xi][yi] == best_ptr )
            done = T;
        }
      }
    }
    q = q->qnext;
  }
  reset_back_track_state( qreset );
  best_ptr->state = VISITED;
  delete_heap( frontier_heap, best_ptr );

  return( best_ptr );
}
```

psearch.c

```
/*
  File:        psearch.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Persistence Search - uses distance to goal and distance to cur-
rent
to determine best frontier.
*/

#include "ga_search.h"
#include <sys/time.h>


/* ***************** psearch ****************************************** */
float psearch( sx, sy, gx, gy, ind_chrom_factor, random_seed, density, node,
ticks )
int sx,sy,gx,gy;    /* position in density array for start & goal */
struct factor_struct ind_chrom_factor;
int random_seed;
int density[16][16];
struct node_rec *node[66][66];
float *ticks;
{
    struct node_rec *current, *previous, *start, *goal;
    struct node_rec *frontier_heap[4096];

    long sec, usec;
    static struct timeval *tvp;
    static struct timezone *tzp;
    static int first = T;
    float dist_traveled = 0.0;
    int k;

    if( first ) {
      tvp = (struct timeval *)malloc(sizeof(struct timeval));
      tzp = (struct timezone *)malloc(sizeof(struct timezone));
      first = F;
    }

    srandom(random_seed);  /*  seed the random generator */
    heap_size = 0;
    make_array(density, node);
    start = find_node(node, sx, sy);
    start->state = START;
```

```
    start->dist_from_start = 0.0;
    goal = find_node(node, gx, gy);
    goal->state = GOAL;

    for (k=0;k<4096;k++)
      frontier_heap[k] = NULL;
    current = start;
    previous = start;
#ifdef IRIS
    /* initialize the IRIS system */
    initialize(PERSTITLE);
#endif
    gettimeofday(tvp, tzp);
    sec = tvp->tv_sec;
    usec = tvp->tv_usec;

    while ( current != goal) {
      if( adjacent(current, goal) ) {
        dist_traveled = dist_traveled + update_dist( current, goal );
        current = goal;
      }
      else {
        p_update_frontier_list( node, current, previous, frontier_heap, start,
                                goal, ind_chrom_factor );
        if( heap_size == 0) {
          printf("\nENDING SEARCH BEFORE GOAL %d %d  - no more frontier",
                 current->xi, current->yi);
          break;
        }
        previous = current;
        current = p_pick_best_frontier( node, current, frontier_heap,
                                        goal, ind_chrom_factor );
        dist_traveled = dist_traveled + current->dist_from_current;
      }
#ifdef IRIS
      draw_terrain(node,start,goal,current,dist_traveled,ind_chrom_factor);
#endif
    }

    gettimeofday(tvp, tzp);
    *ticks = (float)(tvp->tv_sec - sec) + (tvp->tv_usec - usec)/1000000.0;
#ifdef IRIS
    sleep( SLEEPTIME );
#endif
    goal->state = GOAL;
    start->state = START;

    return( dist_traveled );
}
```

```
/* ***************** p_update_frontier_list ************************* */

p_update_frontier_list( node, c, p, frontier_heap, start, goal, factor )
struct node_rec *node[66][66];
struct node_rec *c;         /* current */
struct node_rec *p;         /* previous */
struct node_rec *frontier_heap[4096];
struct node_rec *start;
struct node_rec *goal;
struct factor_struct factor;
{
  int xi, yi, base_xi, base_yi, top_xi, top_yi;
  float old_subtotal;

  base_xi = c->xi - 1;
  base_yi = c->yi - 1;
  top_xi  = c->xi + 1;
  top_yi  = c->yi + 1;

  for (xi=base_xi;xi<=top_xi;xi++)
    for (yi=base_yi;yi<=top_yi;yi++)
      if((node[xi][yi]->state == UNTOUCHED) || (node[xi][yi]->state == GOAL)) {
        node[xi][yi]->state = FRONTIER;
        node[xi][yi]->predecessor = c;
        node[xi][yi]->dist_from_goal  = update_dist( node[xi][yi], goal );
        node[xi][yi]->subtotal =
                           node[xi][yi]->dist_from_goal * factor.goal_dist;
        insert_heap( frontier_heap, node[xi][yi] );
      }
}

/* ***************** p_pick_best_frontier *************************** */
/* Finds best frontier node, returns it */

struct node_rec *p_pick_best_frontier( node, current, frontier_heap, goal, fac-
tor )
struct node_rec *node[66][66];
struct node_rec *current;
struct node_rec *frontier_heap[4096];
struct node_rec *goal;
struct factor_struct factor;
{
  struct node_rec *best_ptr, *q, *qend, *qreset;
  float node_cost = BIG_NUMBER;
  float norm = 1.0; /* (current->dist_from_goal / 16.0);   normalize factor */
  float lower_bound = frontier_heap[0]->subtotal;
  float upper_bound = BIG_NUMBER;
  int xi, yi, k;
  float steps;
```

```
  best_ptr = current;
  q = current;
  qend = current;
  qreset = current;
  current->back_track_state = VISITED;
  current->dist_from_current = 0.0;

  while ( (lower_bound < upper_bound) && (q != NULL) ) {
    steps = q->dist_from_current + 1.0;
    for(k=0;k<8;k++) {
      if (k==4)
        steps =  q->dist_from_current + SQRT2;
      xi = gen_xi( k, q->xi );
      yi = gen_yi( k, q->yi );

      if ( ( (node[xi][yi]->state == VISITED) ||
             (node[xi][yi]->state == FRONTIER) ||
             (node[xi][yi]->state == START) ) &&
           node[xi][yi]->back_track_state == UNTOUCHED ) {
        node[xi][yi]->dist_from_current = steps;
        node[xi][yi]->back_track_state = VISITED;
        node[xi][yi]->qreset = qreset;
        qreset = node[xi][yi];

        if ( (node[xi][yi]->state == VISITED) ||
             (node[xi][yi]->state == START) ) {
          qend->qnext = node[xi][yi];
          qend = node[xi][yi];
        }
        else {    /* node[xi][yi]->state == FRONTIER */
          node_cost = node[xi][yi]->subtotal +
                      node[xi][yi]->dist_from_current * factor.current_dist;

          if (node_cost < upper_bound) {
            upper_bound = node_cost;
            best_ptr = node[xi][yi];
          }
        }
      }
    }
    q = q->qnext;
    lower_bound = frontier_heap[0]->subtotal + steps * factor.current_dist;
  }
  reset_back_track_state( qreset );
  best_ptr->state = VISITED;
  delete_heap( frontier_heap, best_ptr );

  return( best_ptr );
}
```

tsearch.c

```
/*
  File:        tsearch.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    This is a multi-heuristic search that takes in the bias
factors in the form of an eight digit hexadecimal number.
*/

#include "ga_search.h"
#include <sys/time.h>

int heap_size;


/* ***************** search ******************************************** */

float search( sx, sy, gx, gy, ind_chrom_factor, random_seed, density, node,
ticks )
int sx,sy,gx,gy;     /* position in density array for start & goal */
struct factor_struct ind_chrom_factor;
int random_seed;
int density[16][16];
struct node_rec *node[66][66];
float *ticks;
{
    struct node_rec *current, *previous, *start, *goal;
    struct node_rec *frontier_heap[4096];

    long sec, usec;
    static struct timeval *tvp;
    static struct timezone *tzp;
    static int first = T;

    float dist_traveled = 0.0;
    int k;

    if( first ) {
      tvp = (struct timeval *)malloc(sizeof(struct timeval));
      tzp = (struct timezone *)malloc(sizeof(struct timezone));
      first = F;
    }
```

```
    srandom(random_seed);  /*  seed the random generator */
    heap_size = 0;
    make_array(density, node);

    start = find_node(node, sx, sy);
    start->state = START;
    start->dist_from_start = 0.0;
    goal = find_node(node, gx, gy);
    goal->state = GOAL;

    for (k=0;k<4096;k++)
      frontier_heap[k] = NULL;
    current = start;
    previous = start;

#ifdef IRIS
    /* initialize the IRIS system */
    initialize(GATITLE);
#endif

    gettimeofday(tvp, tzp);
    sec = tvp->tv_sec;
    usec = tvp->tv_usec;

    while ( current != goal) {
      if( adjacent(current, goal) ) {
        dist_traveled = dist_traveled + update_dist( current, goal );
        current = goal;
      }
      else {
        update_frontier_list( node, current, previous, frontier_heap, start,
                              goal, ind_chrom_factor );
        if( heap_size == 0) {
          printf("\nENDING SEARCH BEFORE GOAL - no more frontier");
          break;
        }
        previous = current;
        current = pick_best_frontier( node, current, frontier_heap, goal,
                                      ind_chrom_factor );
        dist_traveled = dist_traveled + current->dist_from_current;
      }
#ifdef IRIS
      draw_terrain(node,start,goal,current,dist_traveled,ind_chrom_factor);
#endif
    } /* end while loop */

    gettimeofday(tvp, tzp);
    *ticks = (float)(tvp->tv_sec - sec) + (tvp->tv_usec - usec)/1000000.0;
```

```
#ifdef IRIS
    sleep( SLEEPTIME );
#endif

    goal->state = GOAL;
    start->state = START;

#ifdef SUN
    /* Print to standard output */
    /* not normally used, but optional for sun */
/*
    printf("\n");
    print_node(node);
    printf("\n");
    printf("\nDIST = %f", dist_traveled);
*/
#endif

    return( dist_traveled );
}
```

tfrontier.c

```
/*
  File:         tfrontier.c
  Programmer:   g.b. parker
  Environment:  any
  Language:     C
  Date:          9 july 92
  Revised:
  Comments:     Maintenance of frontier list
*/

#include "ga_search.h"
#include "math.h"


/* ***************** update_frontier_list **************************** */
/* Looks two away from the current node to update stable search
characteristics */

update_frontier_list( node, c, p, frontier_heap, start, goal, factor )
struct node_rec *node[66][66];
struct node_rec *c;          /* current */
struct node_rec *p;          /* previous */
struct node_rec *frontier_heap[4096];
struct node_rec *start;
struct node_rec *goal;
struct factor_struct factor;
{
  int xi, yi, base_xi, base_yi, top_xi, top_yi;
  float old_subtotal;

  base_xi = c->xi ==  1 ?  1 : c->xi - 2;
  base_yi = c->yi ==  1 ?  1 : c->yi - 2;
  top_xi  = c->xi == 64 ? 64 : c->xi + 2;
  top_yi  = c->yi == 64 ? 64 : c->yi + 2;

  update_adjacent_obstacles( node, c );
```

```
  for (xi=base_xi;xi<=top_xi;xi++) {

    for (yi=base_yi;yi<=top_yi;yi++) {

      if ( ((node[xi][yi]->state == UNTOUCHED) ||
            (node[xi][yi]->state == GOAL) )  && adjacent(node[xi][yi],c) ){
        node[xi][yi]->state = FRONTIER;
        node[xi][yi]->predecessor = c;
        node[xi][yi]->dist_from_goal  = update_dist( node[xi][yi], goal );
        node[xi][yi]->dist_from_start  = update_dist( node[xi][yi], start );
        node[xi][yi]->subtotal =
                 compute_subtotal( node[xi][yi], factor,
                                   calc_momentum(node[xi][yi],c,p),
                               update_crowd_sides(node,node[xi][yi]),
                                   update_crowd_diag(node,node[xi][yi]) );
        insert_heap( frontier_heap, node[xi][yi] );
      }

      else if ( node[xi][yi]->state == FRONTIER ) {
        old_subtotal = node[xi][yi]->subtotal;
        node[xi][yi]->subtotal =
                 compute_subtotal( node[xi][yi], factor,
                                   calc_momentum(node[xi][yi],c,p),
                                   update_crowd_sides(node,node[xi][yi]),
                                   update_crowd_diag(node,node[xi][yi]) );
        if ( !equalf(old_subtotal,node[xi][yi]->subtotal) ) {
          move_heap( frontier_heap, node[xi][yi]->frontier_index );
        }
      }
    }
  }
}
```

```
/* ***************** update_adjacent_obstacles ********************** */
/* Records for future use which adjacent nodes are obstacles */

update_adjacent_obstacles( node, c )
struct node_rec *node[66][66];
struct node_rec *c;         /* current */
{
  int xi, yi, base_xi, base_yi, top_xi, top_yi;

  base_xi = c->xi - 1;
  base_yi = c->yi - 1;
  top_xi  = c->xi + 1;
  top_yi  = c->yi + 1;

  for (xi=base_xi;xi<=top_xi;xi++)
    for (yi=base_yi;yi<=top_yi;yi++)
      if ( node[xi][yi]->state == OBSTACLE )
        node[xi][yi]->back_track_state = OBSTACLE;
}



/* ***************** pick_best_frontier ***************************** */
/* finds best frontier node, returns it */

struct node_rec *pick_best_frontier(node,current,frontier_heap,goal,factor)
struct node_rec *node[66][66];
struct node_rec *current;
struct node_rec *frontier_heap[4096];
struct node_rec *goal;
struct factor_struct factor;
{
  struct node_rec *best_ptr, *q, *qend, *qreset;
  float node_cost = BIG_NUMBER;
  float norm = 1.0; /* (current->dist_from_goal / 16.0);   normalize factor */
  float lower_bound = frontier_heap[0]->subtotal;
  float upper_bound = BIG_NUMBER;
  int xi, yi, k;
  float steps;

  best_ptr = current;
  q = current;
  qend = current;
  qreset = current;
  current->back_track_state = VISITED;
  current->dist_from_current = 0.0;
```

```
  while ( (lower_bound < upper_bound) && (q != NULL) ) {

    steps = q->dist_from_current + 1.0;
    for(k=0;k<8;k++) {
      if (k==4)
        steps =  q->dist_from_current + SQRT2;
      xi = gen_xi( k, q->xi );
      yi = gen_yi( k, q->yi );

      if ( ( (node[xi][yi]->state == VISITED) ||
             (node[xi][yi]->state == FRONTIER) ||
             (node[xi][yi]->state == START) ) &&
            node[xi][yi]->back_track_state == UNTOUCHED ) {
        node[xi][yi]->dist_from_current = steps;
        node[xi][yi]->back_track_state = VISITED;
        node[xi][yi]->qreset = qreset;
        qreset = node[xi][yi];

        if ( (node[xi][yi]->state == VISITED) ||
             (node[xi][yi]->state == START) ) {
          qend->qnext = node[xi][yi];
          qend = node[xi][yi];
        }
        else {     /* node[xi][yi]->state == FRONTIER */
          node_cost = node[xi][yi]->subtotal +
                  node[xi][yi]->dist_from_current * factor.current_dist +
                 calc_move_away(node[xi][yi],current,goal) * factor.move_away;
          if (node_cost < upper_bound) {
            upper_bound = node_cost;
            best_ptr = node[xi][yi];
          } /* end if */
        } /* end else */
      } /* end if */
    } /* end for loop */
    q = q->qnext;
    lower_bound = frontier_heap[0]->subtotal + steps * factor.current_dist;
  } /* end while loop */

  reset_back_track_state( qreset );
  best_ptr->state = VISITED;
  delete_heap( frontier_heap, best_ptr );

  return( best_ptr );
}
```

```
/* ***************** reset_back_track_state ************************* */
/* Resets node record fields used to perform the backtrack search */

reset_back_track_state( qreset )
struct node_rec *qreset;
{
  struct node_rec *temp;

  while (qreset != NULL) {
    temp = qreset->qreset;
    qreset->back_track_state = UNTOUCHED;
    qreset->qreset = NULL;
    qreset->qnext = NULL;
    qreset = temp;
  }
}


/* ***************** update_list ************************************* */
/* Euclidean distance between input nodes */

float update_dist( n1, n2 )
struct node_rec *n1;
struct node_rec *n2;
{
  float x = n1->x - n2->x;
  float y = n1->y - n2->y;

  return( sqrt( x*x + y*y ) );
}


/* ***************** diff_int **************************************** */
/* Absolute difference between two integers */

diff_int( a, b)
int a, b;
{
  int c = a - b;

  if ( c < 0 )
    return( -c );
  else
    return( c );
}
```

```
/* ****************** adjacent ****************************************** */
/* Returns T if the input nodes are adjacent */

adjacent(n1,n2)
struct node_rec *n1;
struct node_rec *n2;
{
  return ( (diff_int(n1->xi,n2->xi) < 2) && (diff_int(n1->yi,n2->yi) < 2) );
}


/* ****************** update_crowd_sides ****************************** */
/* Counts the known adjacent horizontal/vertical obstacles to the frontier
node */

update_crowd_sides(node, f)
struct node_rec *node[66][66];
struct node_rec *f;  /* frontier */
{
  int s_count = 0;

  /* N */
  if ( node[f->xi][f->yi + 1]->back_track_state == OBSTACLE )
    s_count = s_count + 1;
  /* E */
  if ( node[f->xi + 1][f->yi]->back_track_state == OBSTACLE )
    s_count = s_count + 1;
  /* S */
  if( node[f->xi][f->yi - 1]->back_track_state == OBSTACLE )
    s_count = s_count + 1;
  /* W */
  if( node[f->xi - 1][f->yi]->back_track_state == OBSTACLE )
    s_count = s_count + 1;
  return( s_count );
}
```

```
/* ***************** update_crowd_diag ****************************** */
/* Counts the known adjacent diagonal obstacles to the frontier node */

update_crowd_diag(node, f)
struct node_rec *node[66][66];
struct node_rec *f;  /* frontier */
{
  int d_count = 0;

  /* NE */
  if ( node[f->xi + 1][f->yi + 1]->back_track_state == OBSTACLE )
    d_count = d_count + 1;
  /* SE */
  if ( node[f->xi + 1][f->yi - 1]->back_track_state == OBSTACLE )
    d_count = d_count + 1;
  /* SW */
  if( node[f->xi - 1][f->yi - 1]->back_track_state == OBSTACLE )
    d_count = d_count + 1;
  /* NW */
  if( node[f->xi - 1][f->yi + 1]->back_track_state == OBSTACLE )
    d_count = d_count + 1;
  return( d_count );
}


/* ***************** calc_move_away ********************************** */
/* Determines if a move to the frontier would be moving away from the
goal.  Each axis move away counts as two. */

calc_move_away( f, c, g)
struct node_rec *f;
struct node_rec *c;
struct node_rec *g;
{
   int ma_count = 0;

   if( diff_int(f->xi,g->xi) > diff_int(c->xi,g->xi) )
     ma_count = 2;
   if( diff_int(f->yi,g->yi) > diff_int(c->yi,g->yi) )
     ma_count = ma_count + 2;

    return( ma_count );
}
```

```
/* ***************** calc_momentum *********************************** */
/* Returns 0 if no change in direction, 1 if 45 degree change, two if 90
degree change, and three if 135 degree change or node not adjacent */

calc_momentum( f, c, p )
struct node_rec *f;
struct node_rec *c;
struct node_rec *p;
{
  if( adjacent(f,c) && adjacent(c,p) )
    return( diff_int(p->xi - c->xi,c->xi - f->xi) + diff_int(p->yi - c->yi,c-
>yi - f->yi) );
  else
    return( 3 );
}


/* ***************** compute_subtotal ******************************** */
/* Computes the frontier nodes subtotal value dependent on the stable heuris-
tics */

float compute_subtotal( n, factor, m, cs, cd )
struct node_rec *n;        /* the node */
struct factor_struct factor;
int m;                     /* momentum */
int cs;                    /* crowding_sides */
int cd;                    /* crowding_diagonals */
{
  return( n->dist_from_start * factor.start_dist +
          n->dist_from_goal * factor.goal_dist +
          cs * factor.crowd_sides +
          cd * factor.crowd_diag +
          m * factor.momentum );
}
```

theap.c

```
/*
  File:        theap.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Frontier heap functions
*/

#include "ga_search.h"

int heap_size;


/* ***************** insert_heap ************************************* */
/* Inserts a node into the frontier heap */

insert_heap( fh, n )
struct node_rec *fh[4096];  /* frontier_heap */
struct node_rec *n;         /* node to insert */
{
  n->state = FRONTIER;
  n->frontier_index = heap_size;
  fh[heap_size] = n;
  heap_size = heap_size + 1;
  move_heap( fh, heap_size-1 );
}


/* ***************** delete_heap ************************************* */
/* Deletes a node from the frontier heap */

delete_heap( fh, n )
struct node_rec *fh[4096];  /* frontier_heap */
struct node_rec *n;         /* node to delete */
{
  heap_size = heap_size - 1;
  fh[n->frontier_index] = fh[heap_size];
  fh[n->frontier_index]->frontier_index = n->frontier_index;
  n->state = VISITED;
  fh[heap_size] = NULL;
  if ( n->frontier_index != heap_size )
    move_heap( fh, n->frontier_index );
  n->frontier_index = NA;
}
```

```
/* ***************** move_heap **************************************** */
/* Moves a node in the frontier heap if required; dependent on the value
of the nodes subtotal field. */

move_heap( fh, i )
struct node_rec *fh[4096];  /* frontier_heap */
int i;                      /* index of node to possibly move */
{
  int parent = (i - 1) / 2;
  int child  = ((2*i+1) >= heap_size) ? i : 2*i+1;
  int child2 = ((2*i+2) >= heap_size) ? i : 2*i+2;
  if ( (child2 != i) && (fh[child2]->subtotal < fh[child]->subtotal) )
    child = child2;
  while (fh[i]->subtotal < fh[parent]->subtotal) {
    swap( fh, i, parent );
    i = parent;
    parent = (i - 1) / 2;
  }
  while (fh[i]->subtotal > fh[child]->subtotal) {
    swap( fh, i, child );
    i = child;
    child  = ((2*i+1) >= heap_size) ? i : 2*i+1;
    child2 = ((2*i+2) >= heap_size) ? i : 2*i+2;
    if ( (child2 != i) && (fh[child2]->subtotal < fh[child]->subtotal) )
      child = child2;
  }
}


/* ***************** swap ********************************************** */
/* Swaps nodes in the frontier heap */

swap( fh, i1, i2 )
struct node_rec *fh[4096];  /* frontier_heap */
int i1, i2;                 /* indexes of nodes to swap */
{
  struct node_rec *temp_ptr = fh[i1];

  fh[i1]->frontier_index = i2;
  fh[i2]->frontier_index = i1;
  fh[i1] = fh[i2];
  fh[i2] = temp_ptr;
}
```

evolve.c

```
/*
  File:        evolve.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Performs selection, crossover, and mutation on input population.
*/

#include "ga_search.h"


/* ***************** evolve ******************************************* */
/* takes in a population, with fitness information which is used to produce
the next population */

evolve( individual, rs )
struct individual_struct *individual[32];
int rs;       /* random_number */
{
  static struct individual_struct *temp_ind[32];
  static int first = T;  /* T or F */
  int k;
  int top = 0;
  int next_spot = 2;
  int mate_heap[31];
  int even = T;

  if ( first ) {
    create_population( temp_ind );
    first = F;
  }
  temp_ind[0]->fit_sum = 0.0;
  temp_ind[1]->fit_sum = 0.0;

  create_mate_heap( mate_heap, individual, rs );
```

```
  for (k=0;k<32;k++) {
    top = top + individual[k]->fitness;
    while ( (mate_heap[1] <= top) && (mate_heap[0] > 0) ) {
      temp_ind[next_spot]->chrom.alleles = individual[k]->chrom.alleles;
      temp_ind[next_spot]->previous_index = k;
      if ( even ) {
        next_spot = next_spot + 2;
        even = next_spot < 30 ? T : F;
      }
      else
        next_spot = get_odd();
      pop_mate_heap(mate_heap);
    }
    if ( individual[k]->fit_sum > temp_ind[0]->fit_sum ) {
      set_equal( temp_ind[0], temp_ind[1], temp_ind[0]->previous_index );
      set_equal( individual[k], temp_ind[0], k );
    }
    else if ( (individual[k]->fit_sum > temp_ind[1]->fit_sum) &&
              (individual[k]->chrom.alleles != temp_ind[0]->chrom.alleles) )
      set_equal( individual[k], temp_ind[1], k );
  }
  crossover( individual, temp_ind, rs );
  /* mutate done in crossover */
}


/* ***************** create_mate_heap ****************************** */
/* Creates a heap of integers which will be used to stochastically choose
individuals for reproduction */

create_mate_heap( mh, ind, rs )
int mh[31];
struct individual_struct *ind[32];
int rs;       /* random_seed */
{
  int k;
  int total_fit = 0;

  srandom(rs);
  mh[0] = 0;

  for (k=0;k<32;k++)
    total_fit = total_fit + ind[k]->fitness;

  for (k=1;k<31;k++)
    insert_mate_heap( mh, (random() % total_fit) );
}
```

```
/************************** crossover functions ****************************/


/* ***************** allele_crossover ******************************** */
/* Performs crossover of the chromosome at a random allele position */

allele_crossover( ind, temp_ind, k, cross_allele )
struct individual_struct *ind[32];
struct individual_struct *temp_ind[32];
int k, cross_allele;
{
  ind[k]->chrom.alleles = temp_ind[k]->chrom.alleles;
  ind[k+1]->chrom.alleles = temp_ind[k+1]->chrom.alleles;

  switch ( cross_allele )
    {
    case 0:
      ind[k]->chrom.factor.start_dist = temp_ind[k+1]->chrom.factor.start_dist;
      ind[k+1]->chrom.factor.start_dist = temp_ind[k]->chrom.factor.start_dist;
    case 1:
      ind[k]->chrom.factor.goal_dist = temp_ind[k+1]->chrom.factor.goal_dist;
      ind[k+1]->chrom.factor.goal_dist = temp_ind[k]->chrom.factor.goal_dist;
    case 2:
      ind[k]->chrom.factor.current_dist =
                                 temp_ind[k+1]->chrom.factor.current_dist;
      ind[k+1]->chrom.factor.current_dist =
                                 temp_ind[k]->chrom.factor.current_dist;
    case 3:
      ind[k]->chrom.factor.crowd_sides =
                                 temp_ind[k+1]->chrom.factor.crowd_sides;
      ind[k+1]->chrom.factor.crowd_sides =
                                 temp_ind[k]->chrom.factor.crowd_sides;
    case 4:
      ind[k]->chrom.factor.crowd_diag = temp_ind[k+1]->chrom.factor.crowd_diag;
      ind[k+1]->chrom.factor.crowd_diag = temp_ind[k]->chrom.factor.crowd_diag;
    case 5:
      ind[k]->chrom.factor.move_away = temp_ind[k+1]->chrom.factor.move_away;
      ind[k+1]->chrom.factor.move_away = temp_ind[k]->chrom.factor.move_away;
    case 6:
      ind[k]->chrom.factor.momentum = temp_ind[k+1]->chrom.factor.momentum;
      ind[k+1]->chrom.factor.momentum = temp_ind[k]->chrom.factor.momentum;
    }
}
```

```
/* ***************** bit_crossover ********************************** */
/* Performs crossover of an allele at a random bit position */

bit_crossover( ind, temp_ind, k, cross_allele )
struct individual_struct *ind[32];
struct individual_struct *temp_ind[32];
int k, cross_allele;
{
  int cross_bit, inv_cross_bit;

  cross_bit = get_mask( rand5() );
  inv_cross_bit = cross_bit ^ MASK4;

  switch ( cross_allele )
    {
    case 1:
      ind[k]->chrom.factor.start_dist =
         ( temp_ind[k]->chrom.factor.start_dist & cross_bit )    |
         ( temp_ind[k+1]->chrom.factor.start_dist & inv_cross_bit );
      ind[k+1]->chrom.factor.start_dist =
         ( temp_ind[k+1]->chrom.factor.start_dist & cross_bit )    |
         ( temp_ind[k]->chrom.factor.start_dist & inv_cross_bit );
      break;
    case 2:
      ind[k]->chrom.factor.goal_dist =
        ( temp_ind[k]->chrom.factor.goal_dist & cross_bit )    |
        ( temp_ind[k+1]->chrom.factor.goal_dist & inv_cross_bit );
      ind[k+1]->chrom.factor.goal_dist =
        ( temp_ind[k+1]->chrom.factor.goal_dist & cross_bit )    |
        ( temp_ind[k]->chrom.factor.goal_dist & inv_cross_bit );
      break;
    case 3:
      ind[k]->chrom.factor.current_dist =
        ( temp_ind[k]->chrom.factor.current_dist & cross_bit )    |
        ( temp_ind[k+1]->chrom.factor.current_dist & inv_cross_bit );
      ind[k+1]->chrom.factor.current_dist =
        ( temp_ind[k+1]->chrom.factor.current_dist & cross_bit )    |
        ( temp_ind[k]->chrom.factor.current_dist & inv_cross_bit );
      break;
    case 4:
      ind[k]->chrom.factor.crowd_sides =
        ( temp_ind[k]->chrom.factor.crowd_sides & cross_bit )    |
        ( temp_ind[k+1]->chrom.factor.crowd_sides & inv_cross_bit );
      ind[k+1]->chrom.factor.crowd_sides =
        ( temp_ind[k+1]->chrom.factor.crowd_sides & cross_bit )    |
        ( temp_ind[k]->chrom.factor.crowd_sides & inv_cross_bit );
      break;
```

```
    case 5:
      ind[k]->chrom.factor.crowd_diag =
        ( temp_ind[k]->chrom.factor.crowd_diag & cross_bit )     |
        ( temp_ind[k+1]->chrom.factor.crowd_diag & inv_cross_bit );
      ind[k+1]->chrom.factor.crowd_diag =
        ( temp_ind[k+1]->chrom.factor.crowd_diag & cross_bit )     |
        ( temp_ind[k]->chrom.factor.crowd_diag & inv_cross_bit );
      break;
    case 6:
      ind[k]->chrom.factor.move_away =
        ( temp_ind[k]->chrom.factor.move_away & cross_bit )     |
        ( temp_ind[k+1]->chrom.factor.move_away & inv_cross_bit );
      ind[k+1]->chrom.factor.move_away =
        ( temp_ind[k+1]->chrom.factor.move_away & cross_bit )     |
        ( temp_ind[k]->chrom.factor.move_away & inv_cross_bit );
      break;
    case 7:
      ind[k]->chrom.factor.momentum =
        ( temp_ind[k]->chrom.factor.momentum & cross_bit )     |
        ( temp_ind[k+1]->chrom.factor.momentum & inv_cross_bit );
      ind[k+1]->chrom.factor.momentum =
        ( temp_ind[k+1]->chrom.factor.momentum & cross_bit )     |
        ( temp_ind[k]->chrom.factor.momentum & inv_cross_bit );
    }
}

/* ***************** crossover **************************************** */
/* The main function */

crossover( individual, temp_ind, rs )
struct individual_struct *individual[32];
struct individual_struct *temp_ind[32];
int rs;      /* random_seed */
{
  int k;
  int cross_allele;

  srandom(rs);
  cross_allele = rand8();

  set_equal( temp_ind[0], individual[0], temp_ind[0]->previous_index );
  set_equal( temp_ind[1], individual[1], temp_ind[1]->previous_index );

  for(k=2;k<32;k=k+2) {
    allele_crossover( individual, temp_ind, k, cross_allele );
    bit_crossover( individual, temp_ind, k, cross_allele );
    mutate ( individual, k, rs );
  }
}
```

```
/* ***************** get_mask ****************************************** */
/* called by bit_crossover */

int get_mask( rn )
int rn;  /* random number */
{
  switch ( rn )
    {
    case 0:
      return MASK0;
    case 1:
      return MASK1;
    case 2:
      return MASK2;
    case 3:
      return MASK3;
    case 4:
      return MASK4;
    }
}


/* ***************** mutate ******************************************** */
/* Runs through each bit of the chromosome determining if if will invert */

mutate ( ind, k, rs )
struct individual_struct *ind[32];
int k, rs;
{
  unsigned int mut_factor1 = 0xfffffff0;
  unsigned int mut_factor2 = 0xfffffff0;
  int g;

  for (g=0;g<28;g++) {
    mut_factor1 = ( mut_factor1 << 1 ) + one_if_mutate();
    mut_factor2 = ( mut_factor2 << 1 ) + one_if_mutate();
  }
  ind[k]->chrom.alleles = ind[k]->chrom.alleles ^ mut_factor1;
  ind[k+1]->chrom.alleles = ind[k+1]->chrom.alleles ^ mut_factor2;
}
```

```
/* ***************** one_if_mutate *********************************** */
/* Returns 1 if mutation is to take place at the present bit */

one_if_mutate()
{
  if ( rand10000() < PROB_BIT_MUTATE )
    return( 1 );
  else
    return( 0 );
}


/* ***************** get_odd ***************************************** */
/* Determines placement of selected individual for reproduction.
Distributes individuals to avoid mating of like chromosomes. */

/* Definitions only pertinent to this function */
#define LOW  0
#define MED  1
#define HIGH 2

get_odd()
{
  static int next = LOW;
  static int base = 1;

  switch ( next )
    {
    case LOW:
      next = MED;
      base = base + 2;
      return( base );
    case MED:
      next = HIGH;
      return( base + 10 );
    case HIGH:
      next = LOW;
      if ( base == 11 ) {
        base = 1;
        return( 31 );
      }
      else
        return( base + 20 );
    }
}
```

```
/* ***************** set-equal ***************************************** */
/* Sets one individual equal to another */

set_equal( from_ind, to_ind, k )
struct individual_struct *from_ind, *to_ind;
int k;
{
  to_ind->chrom.alleles = from_ind->chrom.alleles;
  to_ind->fitness = from_ind->fitness;
  to_ind->fit_sum = from_ind->fit_sum;
  to_ind->previous_index = k;
}
```

eheap.c

```
/*
  File:         eheap.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Frontier heap functions
*/

#include "ga_search.h"


/* ***************** insert_mate_heap ******************************** */

insert_mate_heap( mh, num )
int mh[31];  /* mate_heap */
int num;        /* number to insert */
{
  mh[0] = mh[0] + 1;
  mh[ mh[0] ] = num;
  move_mate_heap( mh, mh[0] );
}


/* ***************** pop_mate_heap ************************************ */
/* Removes top of mate heap */

pop_mate_heap( mh )
int mh[31];  /* mate_heap */
{
  mh[1] = mh[ mh[0] ];
  mh[ mh[0] ] = 0;
  mh[0] = mh[0] - 1;
  move_mate_heap( mh, 1 );
}
```

```
/* ***************** move_mate_heap *********************************** */
/* Readjusts heap after addition/removal of one of its members */

move_mate_heap( mh, i )
int mh[31];     /* mate_heap */
int i;           /* index of num to possibly move */
{
  int parent = i == 1 ? 1 : i / 2;
  int child  = ((2*i) > mh[0]) ? i : 2*i;
  int child2 = ((2*i+1) > mh[0]) ? i : 2*i+1;

  if ( (child2 != i) && (mh[child2] < mh[child]) )
    child = child2;

  while (mh[i] < mh[parent]) {
    swap_num( mh, i, parent );
    i = parent;
    parent = i == 1 ? 1 : i / 2;
  }

  while (mh[i] > mh[child]) {
    swap_num( mh, i, child );
    i = child;
    child  = ((2*i) > mh[0]) ? i : 2*i;
    child2 = ((2*i+1) > mh[0]) ? i : 2*i+1;
    if ( (child2 != i) && (mh[child2] < mh[child]) )
      child = child2;
  }
}
```

```
/* ***************** swap_num ***************************************** */
/* Swaps positions of two members of the mate_heap */

swap_num( mh, i1, i2 )
int mh[31];  /* mate_heap */
int i1, i2;  /* indexes of numbers to swap */
{
  int temp_num;

  temp_num = mh[i1];
  mh[i1] = mh[i2];
  mh[i2] = temp_num;
}
```

tmisc.c

```
/*
  File:        tmisc.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         6 apr 92
  Revised:
  Comments:    Miscellaneous functions
*/

#include "ga_search.h"


/* ***************** gen_xi ******************************************** */
/* Generates an integer value dependent on the input xi and k.
Used with gen_yi to generate all adjacent nodes to (xi,yi). */

int gen_xi( k, xi )
int k;
int xi;
{
  switch (k)
  {
    case 0:
      return( xi );
    case 1:
      return( xi+1 );
    case 2:
      return( xi );
    case 3:
      return( xi-1 );
    case 4:
      return( xi+1 );
    case 5:
      return( xi+1 );
    case 6:
      return( xi-1 );
    case 7:
      return( xi-1 );
    }
}
```

```
/* ***************** gen_yi ******************************************* */
/* Generates an integer value dependent on the input yi and k.
Used with gen_xi to generate all adjacent nodes to (xi,yi). */

int gen_yi( k, yi )
int k;
int yi;
{
  switch (k)
  {
    case 0:
      return( yi+1 );
    case 1:
      return( yi );
    case 2:
      return( yi-1 );
    case 3:
      return( yi );
    case 4:
      return( yi+1 );
    case 5:
      return( yi-1 );
    case 6:
      return( yi-1 );
    case 7:
      return( yi+1 );
    }
}

/* ***************** equalf ******************************************* */
/* Checks if two floats are equal (within 0.0001) */

int equalf( x, y )
float x, y;
{
  if ( ((x-y) < -0.0001) || ((x-y) > 0.0001) )
    return( F );
  else
    return( T );
}
```

```
/* ***************** show_least_nodes ******************************** */
/* Sets the state field to x for all nodes in the shortest path */
/* Not currently used, but available for graphics */

show_least_nodes( node, g )
struct node_rec *node[66][66];
struct node_rec *g;  /* goal */
{
  struct node_rec *best_ptr;
  float best;
  int xi,yi,k;

  while ((g->dist_from_start > 0.0) && (g->dist_from_start < 10000.0)) {
    best = BIG_NUMBER;
    for(k=0;k<8;k++) {
      xi = gen_xi( k, g->xi );
      yi = gen_yi( k, g->yi );
      if( ( node[xi][yi]->state != OBSTACLE) &&
          ( node[xi][yi]->dist_from_start < best ) ) {
        best = node[xi][yi]->dist_from_start;
        best_ptr = node[xi][yi];
      }
    }
    g = best_ptr;
    g->state = X;
  }
}
```

tdisplay.c

```
/*
  File:        tdisplay.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Functions called by all searches to display the search on the
IRIS.  This file should not be excluded from Makefile if compiled on the SUN.
*/

#include "ga_search.h"
#include <gl.h>
#include <device.h>


/* ***************** initialize *************************************** */
/* Initializes graphics systems for output */

initialize(title)
char title[33];
{
             /* set up a preferred size and location for the window */
   prefsize(XMAXSCREEN+1,YMAXSCREEN+1-256);
   prefposition(0,980,0,980);
                      /* open a window for the program */
   winopen("search");
                       /* put a title on the window */
   wintitle(title);
                  /* put the machine into double buffer mode */
   doublebuffer();
                     /* set RGB mode for color */
   RGBmode();
       /* configure the IRIS (means use the above command settings) */
   gconfig();
                    /* queue the redraw device */
   qdevice(REDRAW);
                     /* queue buttons needed */
   qdevice(BUT6);  /* ESC */
   qdevice(BUT50);  /* enter */
   qdevice(BUT4);  /* right shift */
   qdevice(BUT73); /* down arrow */

             /* set the world coordinate system */
   ortho2(-1.0,66.0,-1.0,66.0);
}
```

```
/* ****************** draw_terrain ************************************** */
/* Called by searches to draw the node array */

draw_terrain(node,start,goal,current,dist,chrom)
struct node_rec *node[66][66];
struct node_rec *start,*goal,*current;
float dist;  /* dist_traveled */
struct factor_struct chrom; /* ind_chrom_factor */
{
  short value;
  static int cont = T;
  int mmouse = F;
  int first = T;
  int do_print;

  if( adjacent(start,current) || adjacent(goal,current) )
    cont = T;

  while( (mmouse || first) && cont ) {
    do_print = F;
    draw_grid();
    draw_nodes(node,start,goal,current);

    while( qtest() )
      switch( qread(&value) )
        {
        case BUT6:      /* "ECS" to terminate display for that search */
          cont = F;
          break;
        case BUT50:     /* "return" to halt display */
          mmouse = T;
          break;
        case BUT4:      /* "shift" to continue display */
          mmouse = F;
          break;
        case BUT73:     /* "down arrow" to print node info to standard output
*/
          do_print = T;         /* node is selected by mouse position */
          break;
        default:
          break;
        }
    show_mouse(node,dist,chrom,do_print);

    swapbuffers();     /* change the buffers ... */
    first = F;
  }
}
```

```
/* ****************** show_mouse **************************************** */
/* Shows mouse position and prints node info if selected */

show_mouse(node,dist,c,do_print)
struct node_rec *node[66][66];
float dist;
struct factor_struct c; /* ind_chrom_factor */
int do_print;
{
  int mx_pix = getvaluator(MOUSEX);
  int my_pix = getvaluator(MOUSEY);

  int mx = ((67 * mx_pix)/980) - 1;
  int my = ((67 * my_pix)/980) - 1;

  if ( mx > 65 )
    mx = 65;
  if (my > 65 )
    my = 65;

  RGBcolor(0,0,0);
  square( node[mx][my]->x, node[mx][my]->y, 0.35 );

  if( do_print ) {
    printf("\n dist=%f", dist);
    printf("\n(%d,%d)\n%d state\n%d btstate\n%f subtotal\n%f %d start
            \n%f %d goal\n%f %d current\n%d frontier",
             node[mx][my]->xi, node[mx][my]->yi, node[mx][my]->state,
             node[mx][my]->back_track_state, node[mx][my]->subtotal,
             node[mx][my]->dist_from_start, c.start_dist,
             node[mx][my]->dist_from_goal, c.goal_dist,
             node[mx][my]->dist_from_current, c.current_dist,
             node[mx][my]->frontier_index);
    printf("\n%f from below", node[mx][my]->subtotal -
                          (node[mx][my]->dist_from_start * c.start_dist +
                           node[mx][my]->dist_from_goal * c.goal_dist) );
    printf("\ncs=%d,cd=%d,ma=%d,m=%d\n",
            c.crowd_sides,c.crowd_diag,c.move_away,c.momentum);
  }
}
```

```
/* ***************** draw_nodes *************************************** */
/* Draws node info, can be changed for color */

draw_nodes(node,start,goal,current)
struct node_rec *node[66][66];
struct node_rec *start,*goal,*current;
{
  int xi, yi;

  for(xi=0;xi<=65;xi++)
    for(yi=0;yi<=65;yi++) {
      switch( node[xi][yi]->state )
        {
        case OBSTACLE:
          RGBcolor(0,0,0);
          squaref( node[xi][yi]->x, node[xi][yi]->y, 0.5 );
          break;
        case VISITED:
          /* RGBcolor(0,0,255); */
          circf( node[xi][yi]->x, node[xi][yi]->y, 0.1 );
          circ( node[xi][yi]->x, node[xi][yi]->y, 0.3 );
          break;
        case FRONTIER:
          /* RGBcolor(0,255,0); */
          circ( node[xi][yi]->x, node[xi][yi]->y, 0.3 );
          break;
        }
      }

   /* RGBcolor(255,0,255); */
   circf( current->x, current->y, 0.3 );

   /* RGBcolor(255,0,0); */
   circf( start->x, start->y, 0.4 );
   circf( goal->x, goal->y, 0.4 );
}
```

```
/* ***************** draw_grid ***************************************** */
/* Draws the cross lines for the grid */

draw_grid()
{
  int i;
  float fi;

  /* draw the background color */
  RGBcolor(255,255,255);
  clear();

  RGBcolor(0,0,0);

  for (i=0;i<=66;i++) {
    fi = (float)(i-0.5);
    move2(-0.5,fi);
    draw2(65.5,fi);
    move2(fi,-0.5);
    draw2(fi,65.5);
  }
}


/* ***************** squaref ****************************************** */
/* display filled square for 2D displays */
void squaref(xc,yc,d)
float xc,yc;  /* center point of square */
float d;      /* half of side length */
{
  rectf(xc-d,yc-d,xc+d,yc+d);
}


/* ***************** square ******************************************* */
/* display square for 2D displays */
void square(xc,yc,d)
float xc,yc;  /* center point of square */
float d;      /* half of sidelength */
{
  rect(xc-d,yc-d,xc+d,yc+d);
}
```

tprint.c

```
/*
  File:        tprint.c
  Programmer:  g.b. parker
  Environment: any
  Language:    C
  Date:         9 july 92
  Revised:
  Comments:    Prints to standard output
*/

#include "ga_search.h"

/* ***************** print_density ********************************** */
/* Prints density terrain to standard output */

print_density( density )
int density[16][16];
{
   int i, j;

   for ( j=15; j>=0; j-- ) {
     printf("\n");
     for ( i=0; i<=15; i++ )
       printf("%x ", density[i][j]);
   }
}


/* ***************** print_population ****************************** */
/* Prints the population to standard output */

print_population( i )
struct individual_struct *i[32];
{
  int k;

  for (k=0;k<32;k++)
    printf("\n %d %x %d %f %d", k, i[k]->chrom.alleles, i[k]->fitness, i[k]-
>fit_sum, i[k]->previous_index);
}
```

```
/* ***************** print_node ************************************** */
/* Prints node terrain to standard output */

print_node( node )
struct node_rec *node[66][66];
{
   int i, j;

   for ( j=65; j>=0; j-- ) {
     printf("\n");
     for ( i=0; i<=65; i++ )
       switch (node[i][j]->state)
         {
         case UNTOUCHED:
           printf(".");
           break;
         case OBSTACLE:
           printf("#");
           break;
         case VISITED:
           printf("o");
           break;
         case FRONTIER:
           printf("f");
           break;
         case START:
           printf("S");
           break;
         case GOAL:
           printf("G");
           break;
         case CURRENT:
           printf("O");
           break;
         case SHORTEST:
           printf("s");
           break;
         case X:
           printf("x");
           break;
         }
   }
}
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
Prof. Man-Tak Shing, Code CSSh Computer Science Department Naval Postgraduate School Monterey, CA 93943	4
Prof. Yuh-Jeng Lee, Code CSLe Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
Prof. Kenneth A. DeJong Computer Science Department George Washington University Fairfax, VA 22030	1
Dr. John J. Grefenstette, Code 5514 Navy Center for Applied Research in Artificial Intelligence Naval Research Laboratory Washington, DC 20375-5000	1
Mr. Alan C. Schultz, Code 5514 Navy Center for Applied Research in Artificial Intelligence Naval Research Laboratory Washington, DC 20375-5000	1
USS AMERICA (CV-66) Attn: LCDR Gary B. Parker FPO-AE 09531-2790	2